doi:10.3772/j.issn.1006-6748.2014.04.001

Single-particle 3D reconstruction on specialized stream architecture and comparison with GPGPUs①

Duan Bo (段 勃)②, Wang Wendi, Tan Guangming, Meng Dan
(Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100191, P. R. China)

Abstract

The wide acceptance and data deluge in medical imaging processing require faster and more efficient systems to be built. Due to the advances in heterogeneous architectures recently, there has been a resurgence in the first research aimed at FPGA-based as well as GPGPU-based accelerator design. This paper quantitatively analyzes the workload, computational intensity and memory performance of a single-particle 3D reconstruction application, called EMAN, and parallelizes it on CUDA GPGPU architectures and decouples the memory operations from the computing flow and orchestrates the thread-data mapping to reduce the overhead of off-chip memory operations. Then it exploits the trend towards FPGA-based accelerator design, which is achieved by offloading computing-intensive kernels to dedicated hardware modules. Furthermore, a customized memory subsystem is also designed to facilitate the decoupling and optimization of computing dominated data access patterns. This paper evaluates the proposed accelerator design strategies by comparing it with a parallelized program on a 4-cores CPU. The CUDA version on a GTX480 shows a speedup of about 6 times. The performance of the stream architecture implemented on a Xilinx Virtex LX330 FPGA is justified by the reported speedup of 2.54 times. Meanwhile, measured in terms of power efficiency, the FPGA-based accelerator outperforms a 4-cores CPU and a GTX480 by 7.3 times and 3.4 times, respectively.

Key words: Stream architecture, general purpose graphic processing unit (GPGPU), field programmable gate array (FPGA), cryo-EM

0 Introduction

In high performance computing (HPC) community, the wider applied general purpose graphic processing units (GPGPUs) and the lesser extent field programmable gate arrays (FPGAs) are two of the most important candidates for extending and accelerating the computing power of standard computer architectures. GPGPUs are of interest for several reasons: inexpensive (cost effective), providing high memory bandwidth and parallelism in large-scale. The performance advantage of GPGPUs has been demonstrated in plenty of scientific applications[1]. In contrast, by introducing specialized accelerators, configurable datapath and memory subsystem, FPGAs allow developers to customize the hardware architecture with respect to their applications' characteristics, and therein lies the power efficiency of using application-specific hardware. Limited by the device capacity and the high implementation cost, the traditional usage of FPGAs is largely restricted to either small-scale applications, such as RSA and FIR, or key kernels of applications, such as the molecular dynamics[2]. Recently, due to the advances in the FPGA technologies and the emergence of fast floating-point elementary function libraries[3], there has been a research trend in the design of customized accelerators that leverages the reconfigurability of FPGAs.

In this paper, recent work on accelerating a large-scale scientific application is introduced, called EMAN[4], which is an open source software package for single-particle 3D reconstruction from Cryo-electron microscopy images, on a customized FPGA accelerator card as well as up-to-date GPGPUs. The EMAN is composed of hundreds of time-consuming kernels showing diverse computing features. How to integrate kernel implementations under a unified framework determines

① Supported by the National Basic Research Program of China (No. 2012CB316502), the National High Technology Research and Development Program of China (No. 2009AA01A129), and the National Natural Science Foundation of China (No. 60921002).

② To whom correspondence should be addressed. E-mail: duanbo@ncic.ac.cn

Received on Dec. 15, 2012

the extent to which the application can be accelerated on FPGAs. Moreover, the overall execution time of applications is often dominated by the efficiency of their data access patterns[5]. One of the main complexities for designing customized memory subsystem is that it is needed to support various computing dominated data access patterns and the data sharing across computing 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 the support of explicit and pattern-based memory access functionalities. The memory subsystem can be viewed as both a framework for creating data access patterns and a runtime system that assists the construction of the data flows. The contribution of our work can be summarized as follows:

- The parallelization strategies are evaluated to accelerate a single-particle 3D reconstruction on up-to-date GPGPU architectures. The applied percolation technique greatly improves performance of accessing off-chip memory. The CUDA EMAN on a GTX480 shows a 6 times speedup over a 4-cores Intel Xeon E5520 CPU.
- A coarse-grained stream architecture on FPGAs is proposed for accelerating the 3D reconstruction. The stream architecture is designed based on key observations of kernel classification. Several optimizations that facilitate the mapping between the application and the stream architecture on FPGAs are proposed. In particular, we develop a novel hybrid memory controller featuring the support of explicit and pattern-based data access dominated by computing modules.
- The proposed stream architecture is implemented and evaluated on our customized FPGA accelerator card. The efficiency is justified by the reported 2.54 times speedup over a 4-cores CPU. Compared to the GPU-CUDA based implementation, the customized accelerator improves power efficiency by 3.4 times.

The rest of this paper is organized as follows. The background of single-particle 3D reconstruction and its characterization are introduced in Section 1. Section 2 and Section 3 explain the parallelization and optimization strategies on GPGPU and FPGA architectures respectively. The experimental results are discussed in Section 4, whereas we conclude the work in Section 5.

1 Single-particle 3D reconstruction

The EMAN is a software package designed to handle nearly all aspects of the single-particle 3D reconstruction, such as particle selection, particle alignment and 3D model projection/reconstruction, 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 the amount of particles (images), which are selected from scanned raw micrographs or CCD (charge-coupled detector) frames. A refinement procedure for the final 3D image is a model-based iteration. The preliminary model is used as the starting point for the refinement loop. True convergence is achieved when the model remains unchanged for several successive iterations. The refinement loop begins by generating a set of M projections with uniformly distributed orientations. Then the N particles are classified into different classes by the projections (particle classification) and aligned to generate an average model for each class (class averaging). Finally, the average models with assigned Euler anglesare used to construct a new 3D model for the next round of refinement.

Algorithm 1 illustrates pseudo-code of the refinement algorithm. Generally, the particle classification does an excellent job of assigning particles to the correct class. However, in some cases low signal-to-noise ratio in electron micrographs will lead to miss-assignment for some particles. Fortunately the incorrectly assigned particles will be eliminated by class averaging procedure. The iteration of refinement is a sequential process because current refinement depends on the model generated in the previous iteration. However, abundant of parallelism is observed within most single refinement procedure. Since the operations of classification and average are similar, the same parallelization

Algorithm 1: Reconstruction Alogrithm

```
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              PTFAlign (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
```

can be applied. For simplicity of presentation, only the analysis of classification (line 3-5) is presented. Note that a common procedure of particle classification is rotational and translational alignment (the RTAlign kernel in Algorithm 1), which occupies over 95% of total execution time. Therefore, this paper focuses the discussion on FPGA acceleration of the particle alignment only.

1.1 Workload characterization

More than 90% execution time of the reconstruction program is consumed by the rotational and translational algorithm (RTFAlign). In this section a detailed characterization of this procedure is presented, focusing on computation, memory behavior and parallelism. The workload characterization motives our parallel implementation on a many-core processor (GTX480). In the experiments, 2 images in real applications with typical pixels are used: 256×256, 512×512. Table 1 summarizes the profiling experiments for execution time, arithmetic intensity, branch ratio, working set and cache performance. We present the experimental results for one run of multiple RTFAlign executions in a refine iteration.

1.1.1 Arithmetic intensity

Current multi/many-core technology integrates a large number of arithmetic units into one chip. The number of arithmetic operations of an application should be obviously a critical measure of performance on such a multi-core architecture. Arithmetic intensity is used to show the ratio between arithmetic operations and the number of input and output words required to be computed in a kernel to evaluate how well it is adapted to the many-core architecture. The third column in Table 1 is the arithmetic intensity of each kernel in the RTFAlign algorithm. All but two kernels (dot, Translate) have more than one of arithmetic intensity. The vector dot multiplication's arithmetic intensity is less than 1 because it needs 2 arithmetic operations (one multiplication and one add) and 3 memory operations (two loads and one store). The translation only involves with memory movement. The feature of high arithmetic intensity seems to be well suited to the many-core architecture like GPU. However, note that computations with branch instruction may not utilize many arithmetic units hindered by the serialization of SIMD pipeline in GPU multithread execution. The fourth column in Table 1 is the ratio of branch instruction execution in each kernel. Although several kernels have more than 10% branch instruction execution, an analysis of source codes indicates that most of the branch executions are out of loops, which means the branch instruction does not result in stall of SIMD on GPU. Besides, through orchestrating data-task mapping, most of branch instructions within a loop could be grouped together into one warp of CUDA. Observation1: the RTFAlign algorithm is high arithmetic intensive.

Table 1 Workload characterization for RTFAlign. AI: Arithmetic Intensity. The profiling is collected from a single run of RTFAlign. The time is measured as millisecond.

kernel	size	time	AI	branch	memory	LLC
MCF	256	20.97	8.5	2.4%	1612KB	1.3%
	512	106.3		1.4%	6347KB	1.2%
CCFX	256	1.74	4.3	1.5%	714KB	1.2%
	512	7.16		1.2%	2858KB	1.2%
CCF	256	5.12	6.3	1.2%	524KB	1.5%
	512	20.65		1.8%	2097KB	1.3%
Unwrap	256	3.75	10.5	8.9%	476KB	0.1%
	512	15.79		17.2%	1905KB	0.1%
Rotate	256	3.71	23.5	5.2%	524KB	0.9%
	512	14.18		12.7%	2097KB	2.4%
Translate	256	1.59	0	7.3%	524KB	30%
	512	6.36		15.2%	2097KB	46%
Dot	256	0.28	0.67	1.8%	524KB	2.5%
	512	1.32		3.8%	2097KB	2.7%

1.1.2 Memory

Another feature of the many-core architecture like GPU is configured with small shared on-chip memory. Detailed memory behavior is examined in several ways. First, The working set is measured. As shown in the 6th column in Table 1 the memory usage of all kernels is proportional to square of pixel. Limited by current cryo-EM technology, the pixel of most of images is less than 1024×1024. Thus, optimization for small shared on-chip memory could take the advantage of the feature of small working set. Second, to quantify locality we sample cache access performance. The last column in Table 1 reports the L1 data cache miss rates on an AMD Barcelona processor. Most of kernels has good spatial locality, which is favorable to share data among threads. Note that Translate kernel has high cache miss rate when the pixel size is larger. The translational algorithm needs to translate a 2-D array from the left-bottom to right-top and the memory access strides are proportional to the size of one dimension. Therefore, the access pattern incurs a large number of conflict miss. Observation 2: The instantaneous working set of the RTFAlign algorithm is small and has good spatial locality.

1.1.3 Parallelism

Looking at the reconstruction algorithm in Algorithm 1 we could exploit parallelism at two levels. A coarse-grained parallelism is exploited by partitioning N particles into multiple threads, and then each thread performs rotational and translational alignment for its own particles, which obviously leads to a higher instantaneous working set because the threads are performing RTF Alignin parallel. Thus, contentions of shared on-chip memory and bandwidth become bottlenecks for scalability of the coarse-grained parallel algorithm. Therefore, an alternative way is to exploit a fine-grained parallelism within the rotational and translational algorithm. Observation 3: There is abundant parallelism in the RTFAlign algorithm. A fine-grained parallelism is preferred.

This workload characterization shows that the single-particle 3D reconstruction algorithm is of high parallelism, high arithmetic intensity and high spatial locality. At the first glance, these features are suitable for highly parallel many-core architecture, however, it is not trivially data parallel. For example, strategies to develop a proper data structure for memory management, coalesced memory access and orchestrate data movement with thread mapping need to be carefully designed for high performance.

2 Exploiting parallelism on GPGPUS

All data reside in off-chip global memory with the highest latency (up to 1000 cycles) at the beginning of execution. Although bandwidth to the off-chip memory is very high at more than 100GB/s, it can saturate if all threads request access are contiguous elements of memory, which enables hardware to coalesce memory accesses to the same DRAM page (i.e. contiguous 16-word lines). Therefore, it is needed to apply optimizations that coalesce data accesses and reuse datain order to achieve good performance[6]. Considering the memory hierarchy of CUDA model, we generalize its memory to two levels of memory: off-chip global memory with high latency and on-chip shared memory with low latency. Inspired by previous work on other multi-threaded many-core architecture[7], percolation programming model is extended to enable memory coalesce optimized and hide the off-chip memory latency. Note that a notable feature of GPU-CUDA is massive multi-threading to hide latency on a memory hierarchy with high bandwidth. On the architectural side the latency-hidden is implemented at instruction level. If the threads are partitioned into memory threads and computation ones, a more flexible interface is exposed to programmer for scheduling memory and computation operations. A basic idea of percolation programming is to decouple computation with memory access. The percolation strategy for memory coalesce optimization consists of three concurrent pipelining processes:

- Inward percolation: This step reads data from off-chip global memory to on-chip shared memory. The key insight is to orchestrate mapping between data and threads to avoid un-coalesced access.
- Computation: After the required data reside in the shared memory, parallel threads perform real calculation with loaded data.
- Outward percolation: Finally we write back the results to global memory. Again a thread mapping strategy needs to be selected to improve coalesced access.

There are two remarkable advantages of percolation strategy. First, since the off-chip memory access operations are decoupled with computation, different optimization strategies are developed to the three pipelined stages, respectively. For example, the thread-data mapping may be adaptive through different stages. Second, inward percolation-computation-outward percolation is actually pipelined and the overhead of off-chip memory access is hidden if the pipelining is full. Besides, if a program exhibits data locality, the computation threads also can re-use the data percolated to on-chip memory. Note that the percolation model requires a synchronization at the end of each pipelining step. The tasks of the three steps may be assigned to different thread blocks in CUDA. Unfortunately there is no mechanism to support synchronization among thread blocks, and a global synchronization proposed by Volkov[8] is employed. The idea of percolation is similar to streaming programming style of gather-compute-scatter. The streaming programming uses a DMA mechanism to address issues of data movement between CPU system memory and GPU memory, percolation utilizes the massive multi threaded hardware units to hide the overhead through the GPU memory hierarchy and requires less hardware cost. In this section, it shows how to apply percolation programming to improve coalesced memory access for the alignment algorithm.

2.1 Rotation and translation

The kernels Rotating, Translating and unwrapping in Table 1 are responsible for rotation and shift operations on an image. The basic procedures of the three kernels are similar: for any given point in the new image, calculate its corresponding coordinate in the original image and find the right pixel, then store the points into the new image. Thus the algorithm can be general-

ized into 2 steps: coordinate computation and interpolation. Their difference in coordinate computation, however, affects their memory access behaviors, respectively. For translation operations, the coordinates of pixels are transformed between two rectangular coordinate systems, which leaves the dataloads and stores coalesced easily. Both rotation and unwrap operations transform between rectangular and polar coordinate systems. Their computation however leads to the amount of un-coalesced memory access because of the unaligned coordinate system. Since both operations have the same computational behavior, only the optimization strategy is described to rotation.

For rotation operation parameters determining the angle the pixel needs to move around the image center are given in the polar coordinate. Therefore, to access the corresponding element in memory, a polar to rectangular coordination transformation has to be performed to calculate the actual index of the element. For a point located in (x, y) with a rotation angle of θ, the old point coordination is $(x\cos\theta + y\sin\theta, x\sin\theta + y\cos\theta)$. Since transformation result is unlikely to be right on an integerpoint, we need to perform an interpolation to compute the value of the 2-D function at the given position. Such interpolation involves 4 points around the newpoint. The result is determined by the point's distance to each of its 4 surrounding integer points. However, such pattern violates the coalesce rules directly.

Note that the rotation algorithm is iterated with different angles. Our algorithm performs rotations of multiple angles in parallel. The image is concurrently rotated from three consecutive angles. With the percolation idea the interpolation operation is split into three-stages: read data from the original coordinate, compute, write back to the new coordinate. For read operation the elements in the same row are assigned to the same warp, thus the words accessed by all threads are easy to be arranged into the same segment of size equal to 128 bytes. Then the mapping strategy between data and thread warp for writing back operations is determined by the rotation angle. All elements along the same rotation angle are assigned to the same warp. Therefore, the un-coalesced memory accesses are reduced. On the other hand, such data access pattern clearly demonstrates data locality and reuse. Each iteration reads 4 adjacent points in the image, and the following iteration is expected to be reused between 1 or 2 of the previous loaded points. The exact ratio of data reuse is $|\tan(2\times\theta)|\times 2 + (1 - |\tan(2\times\theta)|) = 1 + |\tan(2\times\theta)|$.

2.2 Correlation function

The calculation of both auto- and cross-correlation functions is composed of three similar steps: i) forward Fourier transformation on two input images. ii) blending operation on the transformed images. iii) backward Fourier transformation on the blent image. Among the three correlation functions, CCFX is relatively different in that its operations are on a per-row basis because of the 1D real Fourier transformation. In this sub-section it presents the parallel algorithm for the blending operation and multiple 1D Fourier transformations.

Assume that the size of a particle is $nx \times ny$, the blending operation reads the particle's data and produces one of the same length. The original implementation needs to read two elements from both input streams and write two elements into the result stream, making memory coalescing difficult. Keeping the percolation idea in mind again, we split the blending operation into three explicit stages: Mapping data from global to shared memory, compute with the loaded data in shared memory, write back result to global memory. The parallel algorithm performs blending operations row-by-row of the image. A trick is to choose different data-thread mapping strategies to avoid uncoalesced access in global memory. During reading/writing data in global memory the number for threads is the same with the number of elements, and each element is consecutively mapped to each thread. The real calculation operations are finished by half number of threads, that is, each thread reads/writes two elements in shared memory.

The algorithm described in Algorithm 1 needs 1D Fourier transformations for rotational alignment (CCFX). Note that the transformation in each row is too small to amortize the CUDA kernel invocation overhead. Fortunately, since rows in the matrixes are stored in a continuous manner, and there is no data dependency between calculations on different rows, different iterations of the blending operation are merged to make batched Fourier transformation. While the correlation kernel reads elements from both ends of the array towards its center, the load is kept from the global memory to the shared memory sequential within each thread block. When nx is not multiple of 16, the rows are paded to align the half warpbase addresses to get the maximum coalesced accesses. After such adjustments, batched Fourier transformation demonstrates an approximated $5X$ speed up against original implementation, and the invocation overhead in the correlation kernel is reduced to about 15% of total runtime.

3 Design of customized accelerator on F-PGAS

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 the system is geared towards architecture support that assists the implementation of each kernel category. The kernels of EMAN will be mapped to dedicated hardware modules on FPGAs: computing modules are used to wrap arithmetic operations; memory modules are used to implement data access as well as related address calculation; stream modules are composed of several computing and memory access modules. In this section, the stream architecture will be discussed in detail, which covers the topic of how to extract and classify candidate kernels from EMAN for acceleration, how to implement each hardware module and how to map a complex computing flow to a stream module.

Table 2 Kernel categorization

Name	Category	Description
MCF	C	Autocorrelation
CCFX	C	Cross correlation on x-dimension
CCF	C	Cross correlation with reference
Unwrap	C	Rectangular to polar coordination
Rotate	C	Rotates image by given angle
Translate	C	Translates image by given offset
DOT	C	Scoring the alignment
Resize	M	Changing image size
Rot180	M	Rotate image by 180°
hFlip	M	Flip images horizontally
Bit Reversal	M	Used in FFT kernels
Transposition	M	Used in the row-column 2D FFT
RTFAlign	S	Aligning with RTAlign and hFlip
RTAlign	S	Aligning with Rotate, Translate CCFX, CCF and DOT
MakeRFP	S	Calculate RFP MCF and unwrap

3.1 Kernel classification

Even accounting for only a small fraction of EMAN, particle alignment contains hundreds of kernels and tens of parameters. However, based on partial evaluation and runtime profiling, the 23 most time-consuming kernels can be classified into 3 categories: computing kernels, memory kernels and stream kernels. Table 2 summarizes the identified kernels in EMAN and their workload descriptions. For simplicity of presentation, some computing kernels in Table 2 are combined, thus the number of computing kernels can be reduced to 7.

Memory: Memory kernels are of limited use in their own right, by which it is meant that they must be combined with computing kernels to deliver on computing 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 profiling the memory address sequences, are implemented with a unified data flow module (DFM). By introducing DFM, it becomes possible for computing modules to do pattern-based data access. The topic of how to extract data access patterns is covered in Section 3.3, whereas Section 3.5 introduces the DFM in detail.

Computing: With respect to loop boundaries, computing kernels are composed of one or more loop statements. Instead of implementing computing kernels 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 DFM. Section 3.4 discusses the kernel implantation in detail.

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

3.2 System architecture

Before introducing the hardware implantation in detail, Fig. 1 gives an overview of the stream architecture. The core of the architecture 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 memory addresses exclusively. A unique feature of the hybrid memory controller is the inclusion of 2 dedicated data flow modules (DFMs), which can be used to implement various data access patterns. Intuitively, the overhead of data transfer can be hidden by overlapping them with the computing. In order to

achieve this overlap, a data pre-fetch unit is integrated in the memory controller, which can be configured to do forward as well as backward pre-fetching at the granularity of image line. Section 3.5 gives more details about the hybrid memory controller.

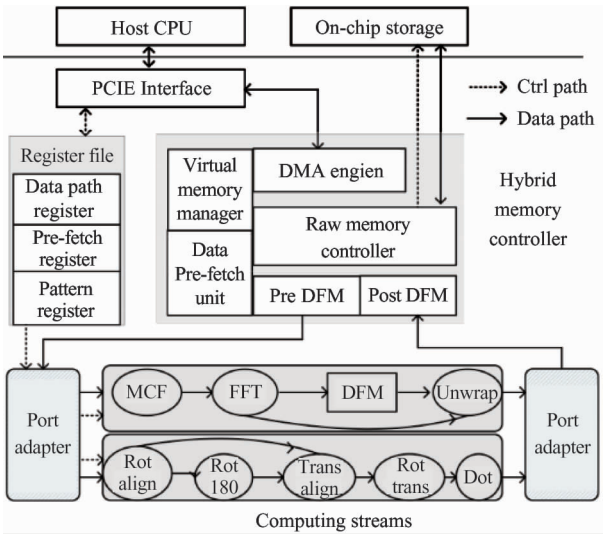

Fig. 1 Overview of system architecture. Functions are implemented in form of a computing stream with kernels mapped as circles. Data flow modules are used to implemented data access patterns

The configurable computing streams are constructed by arranging the hardware modules and bypass channels to form a linear deep pipeline. Fig. 3 illustrates the proposed computing stream to accelerate the RTAlign kernel. Besides the DFMs lie in the memory controller, DFM can also be placed in the computing stream. For example, in Fig. 3, there are two data reorder modules (used for data flow permutations) in the 2D FFT kernel. Multiple computing streams can be placed in parallel, the port adapters are used to switch the data path between the memory controller and the 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 some bypass stages. As a result, a computing stream can be used to implement various functions; 2) By configuring the DFMs and the data per-fetch unit, the memory controller can be controlled to do pattern-based data access. Configuring DFM only needs to read configurable bit file from bit file cache and write it to DFM registers, which takes about fifty nanoseconds. By contrast, FPGAlogic function is traditionally changed by loading new FPGA binary file into FPGA device, which is named function level configuration consuming approximately several milliseconds for Virtex5 LX330.

3.3 Separating computing flow from data flow

Most applications, including EMAN, are rarely developed to take the aforementioned 3 kernel types into consideration. It is a common phenomenon that computing and data access are entangled with each other. The separation between computing flow and data flow manifests itself in two dimensions: First, application-performance is deeply influenced by the ability of overlapping computing with data access[5], by separating-data flow from the computing flow, some advanced data pre-fetching and reordering strategies can be utilized; Second, lots of data access of applications are highly structured, for example, data accesses within loops are often subscripted by loop indices, and result in either sequential or stride data access patterns. The cost to implement computing kernels can be greatly reduced by separating memory operations for dedicated consideration. In this way, computing kernels can be abstracted as black boxes with FIFO in and FIFO out.

The analysis of data access patterns can be simplified and confined within loop statements. A LLVM loop pass is used to facilitate the analysis process. First, data will be clustered into arrays with layout optimization, which makes it possible to use memory addresses to identify data access to specific data types; Second, by recording the memory addresses issued in loop statements, regular and irregular data access patterns can be expressed as fixed-step counters and lookup tables, which map iteration indices to the memory addresses requested by loop statement. In this way, the memory addresses will be calculated statically, and computing kernels can dispense with the overhead of data access and get a simplified view of the computing flow. At last, computing kernels, which contain stateless loops (such as vector addition) only, will be implemented as simple arithmetic modules that operates on a continued data flow.

Furthermore, it is still needed 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 access 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 computing kernels in 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 the memory controller.

3.4 Kernel implementation

Due to the complexity of EMAN, 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 also not a cost effective way to implement some kernels of EMAN on FPGAs, such as 3D-FFTs, heuristic particle selection, etc. Therefore, the performance of the FPGA 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 (Classalign 2) are built. In the following part, the discussion is limited to FPGA acceleration of the RTAlign kernel only. Three representative kernels, mixed radix 2D FFT, image rotation by 180° and the RTAlign stream kernel will be used to demonstrate how to implement each category of kernel in the system.

3.4.1 Computing kernel

The computing kernel implementation is built up on a one-to-one correspondence between the loop statements and the computing modules (circle nodes of Fig. 1). In order to reduce the design cost and increase modular reuse, the computing modules are implemented and wrapped in a unified interface. Given in Fig. 2, the unified interface is composed of standard FIFO signals, flow control signals and signals 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 various complex computing streams.

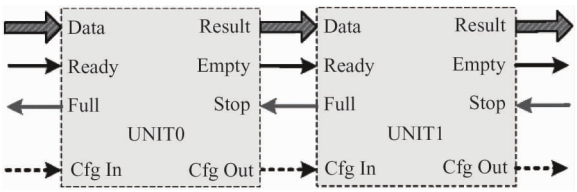

Fig. 2 Interface of computing modules. Forward data transfer and backward flow control are implemented with standard FIFO interface

Based on the unified module interface, an automatic HDL core generator framework in Matlab is developed, which can be used to translate given data flow graphs (DFGs) of kernels (generated by GCC) to HDL implementations. The nodes of DFGs are scheduled to different pipeline stages with respect to data dependency. The pipelined DFG will be used to generate the HDL descriptions. The strength of the proposed core generator is justified by the ability to generate the prime-radix FFT cores in our previous work[9]. There are plenty of existing researches related to FFT design on FPGAs; however the issues of prime-radix FFT, 2D FFT and real FFT are rarely discussed. Therefore, it is still needed to implement the FFT kernels on our own efforts.

The datapath of the computing stream can be controlled by configuring the bypass switches in Fig. 3. Therefore, different FFT factorizations will be mapped to different paths flow through the computing stream. For example, the lower part of Fig. 3 gives the structure of an 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 datapath switching, modules can be shared across configurations. It is possible that with proper organization of the computing flow, the frequency of reconfiguration can be reduced or even avoided.

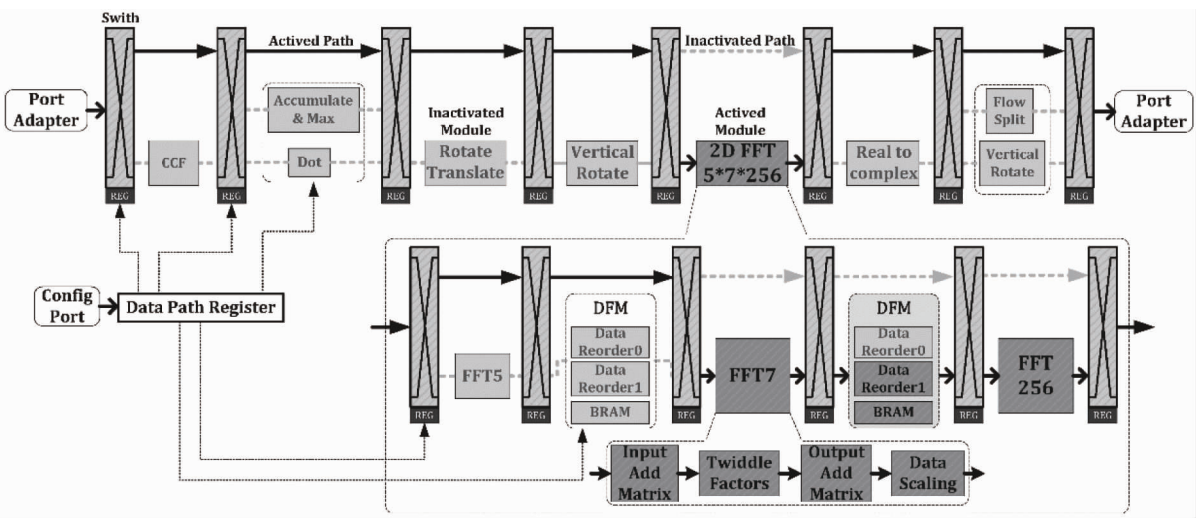

Fig. 3 Computing stream for accelerating the RTAlign kernel

3.4.2 Memory kernel

In order to improve the alignment sensitivity, input images will be compared with up to 4 mirror images (original, horizontal flip, vertical flip and rotating by 180°) in EMAN. For CPU implementation, these mirror images are generated with explicit memory copies. Taking the Rot180 kernel in Table 2 as an example, the memory copies introduce lots of data cache misses and exert severe impact on application performance. With the support of the customized memory controller, the Rot180 kernel can be implemented in following 3 steps: Backward pre-fetching image line-by-line; Buffering data lines; Flushing the buffered lines 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.

Along with the coarse-grained image-level data access patterns, the memory kernels are also extracted and developed for implementing fine-grained data access patterns embedded in the computing kernels. These fine-grained memory access kernels can be utilized to implement various data flow permutations, such as the (de-) interleave and padding/clip operations listed in Table 3.

Table 3 Computing steps of the RTAlign kernel

#	Function	Activated Modules	Memory Controller[1]
1	1D FFT	2D FFT	I: interleave/ O: deinterleave
2	1D IFFT/ CCFX/Acc.	2D FFT/CCF/ Acc. &Max	I: interleave/2-Op.[2]
3	Rotate	RotateTranslate	I:clip/random access
4	1D FFT	2D FFT	I: interleave/ O: column write/ deinterleave
5	1D FFT	2D FFT/Post-Vertical Rotate	O: column write/ deinterleave
6	CCF	CCF	I: interleave/2-Op.[2] O: column write
7	1D IFFT	2D FFT/Pre-Vertical Rotate	O: column write
8	1D IFFT	R2C	I: interleave
9	1D IFFT	2D FFT/Flow Split	
10	MAX	Acc. & Max	I: clip
11	translate		I: clip
12	Dot	Dot	I: interleave/2-Op.[2]

[1] I for input, O for output memory access patterns.

[2] Reading two operands from two separated addresses.

3.4.3 Stream kernel

Fig. 3 illustrates the computing stream for accelerating the RTAlign kernel. The computing and memory access modules are separated by runtime configurable bypass switches. Different functions can be fulfilled by activating the required modules while bypassing the others. The data flow modules (DFM) in the 2D FFT kernel are used to implement 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 switching configuration are wrapped in rewritable registers.

For complex computing flows, which cannot be implemented with a single datapath configuration, the function can be emulated by means of activating the computing stream multiple times with different datapath each time. For example, it is needed to configure the stream pipeline 7 and 12 times to implement the MakeRFP and the RTAlign kernels. Table 3 explains the 12 computing steps of the RTAlign kernel, in which the 2D FFT/IFFT is built upon the row-column algorithm and requires invoking the 1D FFT module twice.

3.5 Memory subsystem and the configurable dataflow module

Our memory system is composed of DRAM, SDRAM and on-chip BRAMs and is managed under a single virtual address space. From the viewpoint of the high-level users, data access on each type of memory can be controlled explicitly by invoking write/read actions at the corresponding memory addresses. The configuration of the virtual memory space is controlled by software and can be changed online. In a typical configuration, the lower addresses (512kB) are allocated to on-chip BRAM followed by SDRAM (16MB) and DRAM (2GB). The on-chip BRAMs are used as a scratch pad memory to store small intermediate data (scalar and vector) between consecutive pipeline stages. The SDRAM is further divided into two parts, which are used for data pre-fetching and intermediate results buffering respectively. The final results will be offloaded to the off-chip DRAM and transferred back to host CPU via DMA through PCIe. The bandwidth of the off-chip DRAM and SDRAM is 3.2GB and 6.4GB, while the system bottleneck lies in the PCIe interface, which is only 2GB.

The pattern-based data access is implemented with the data flow modules (DFMs), which is composed of on-chip BRAMs, address generator, configurable counter and flow control logic. As illustrated in Fig. 4, the 4 BRAMs are organized into 2 groups and are 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 buffered in previous phase. The data rate between input and output of the computing stream may become unmatched, implemented by monitoring the back-pressure signals of the computing stream, a backward flow control mechanism is provided.

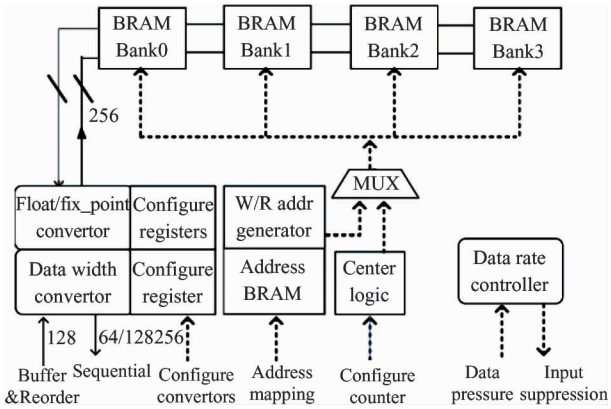

Fig. 4 Data flow module with configurable address generation, data format conversion, configurable counter and flow control logic

The DFMs as well as the data pre-fetching unit in the memory controller are controlled by candidate data access patterns. In particular, DFMs can be controlled by varying address generation logic, data width, port number and the data format converter. For example, data access patterns of fixed steps (regular patterns) will be implemented with the counter logic by setting the counter range and step. In contrast, the address mapping arrays (irregular patterns) will be used to configure the address BRAM in the address generator. Take the computing steps to implement the RTAlign kernel in Table 2 as an example, DFMs can be used to carry out various data access patterns. The process of interleaving two lines of data with length of N can be implemented as follows: 1) Configuring the data pre-fetched unit to read two lines of data each time; 2) Storing the pre-fetched data in a BRAM group in DFM; 3) The address BRAM in the address generator, which is loaded with the interleave $N \times 2$ content in advance, is used to generate reading addresses in the BRAM group.

Along with the 4 address contents illustrated in Fig. 5, 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 relatively 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 15.312kB.

The ability to do column-oriented writing is the major factor that contributes to the performance speed-up of our system. However, the datapath of the DRAM (128bit) and SDRAM (256bit) are both wider than current computing datapath (64bit). The DFM is used, on one hard, to mitigate the bandwidth differences by buffering multiple columns before offloading data to DRAM and SDRAM. On the other hand, the DFM is also used to generate the column-oriented writing addresses, which can be easily mapped to the counter logic.

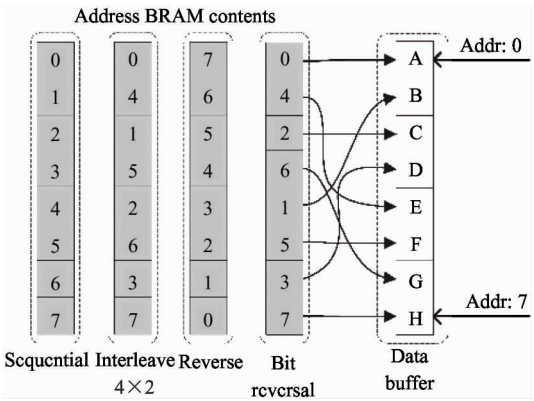

Fig. 5 Address mapping in DFM. The address contents are calculated off-line, are used to do indirect memory accesses

4 Experiment results

4.1 System and experiment setup

The baseline CPU EMAN program is parallelized on a 4-core 2.27GHz Intel Xeon E5520 and compiled with the latest Intel compiler. The proposed GPU algorithm is implemented by rewriting the computing kernels of EMAN using CUDA. The experimental platform is GTX480 (Fermi). 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 64kB local storage, which can be configured as 16kB L1-cache and 48kB shared memory. All SMs share a unified 768KB L2-cache and the 3GB global memory.

The proposed stream architecture is implemented on our customized FPGA accelerator card. Along with the off-chip memory chips (DDR2 SO-DIMM, DDR2 memory chips and QDRII SRAM), the accelerator card is composed of two FPGA chips and communicates with the host CPU via PCIe interface. The computing stream is implemented on the computing node (Xil-

inxXC5VLX330-1), while controlling and data transferring are offloaded to the control node (Xilinx XC5VLX70T). The performance and area of system design on computing node are evaluated with the Xilinx ISE 11.4[10] tool chain. Full system power is measured with the FlukeNorma 4000 Power Analyzer[11].

In the experiments, two classes of images with different resolution are used. Table 4 summarizes their features. The size of pixel is chosen as a metric of problem size in our experiments. For the two data sets, SMALL is a collection of Hepatitis B virus images, while LARGE represents a set of images with larger pixels.

4.2 Kernel performance

Fig. 6(a) and (b) compare the execution time of the computing kernels in Table 2. The speedup of utilizing FPGA varies from 2 times (Max) to 12times (CCFX). It is observed that kernels can achieve different speedups even with a similar computing flow. For example, the Rotate and Unwrap kernel 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 the Rotate kernel.

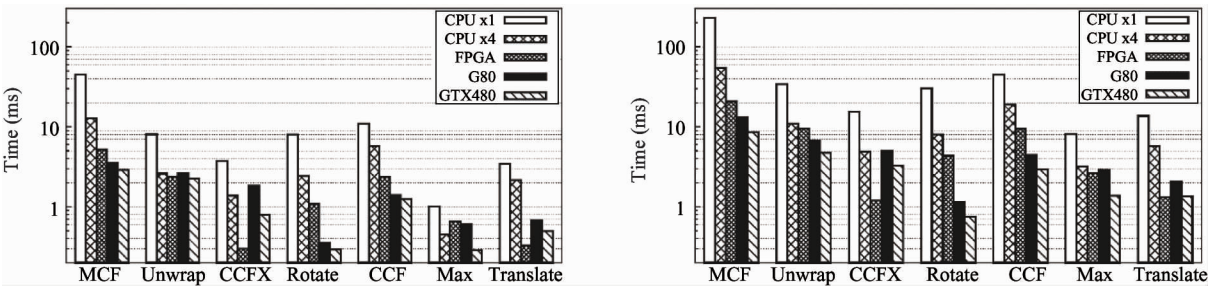

Fig. 6 Kernel execution time of single-threaded CPU, 4-cores CPU, FPGA, G80 and GTX480

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 GPGPUs 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 GPGPUs, which is optimized for sequential data access, incurs a high performance penalty for irregular data access patterns in kernels such as Translate. The speedup of using OpenMP to parallelize EMAN on CPU lies in the range of 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.

4.3 Application speedup

Currently single-precision floating operations can satisfy most of real experimental requirement (though double-precision is necessary in the near future). The algorithms running on GPU-CUDA produce identical results as on CPU for all test sets. In the experiment with Hepatitis B virus (referred to as LARGE in the context) Fig. 7 shows five steps of 3D reconstruction on GPU-CUDA, which are exactly the same with that on CPU.

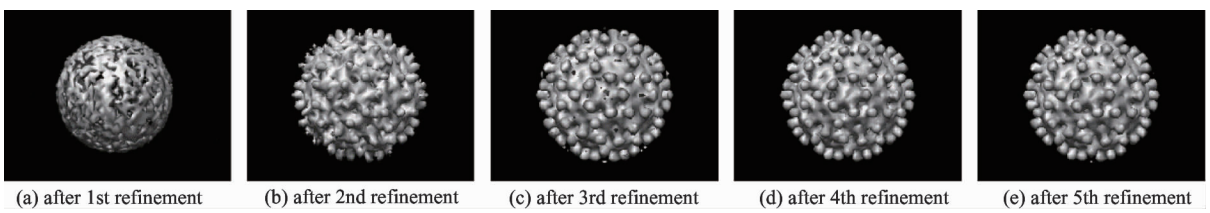

Fig. 7 The first five steps of 3D reconstruction for Hepatitis B virus (LARGE dataset)

Table 5 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 implementations, the total execution time increases about 3 times when image size changs from 256^2 to 512^2, the improved speedup over CPU is largely attributed to the performance degradation on CPU. In contrast, large images can easily saturate the 448 cores in GTX480, which accounts for 2 times speedup. It is considered that FPGA and GPGPUs are more beneficial for Cryo-EM 3D reconstruction on large images.

Table 4 The images used in our experiments

Name	Pixels	Particles	Projections
SMALL	256 × 256	19841	226
LARGE	512 × 512	7224	207

Table 5 Execution time (ms) and speedup normalized to the 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
GTX480	5.82	13.9	13.10	28.8
FPGA	12.31	6.6	49.26	7.7

4.4 Power, area and frequency

Table 6 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 the following power analysis. When measuring the power of system configuration with either FPGA or GPU card, in order to get an accurate evaluation, the power consumption of the host PC is subtracted. For simplicity, it is assumed that the host PC remains idle during the execution of the accelerator cards. The power cost measured in millijoule is computed as the product of working power and the execution time. The multi-threaded EMAN introduces a 3 times speedup at the cost of 27% increased working power; however, with reduced total execution time, the power cost of using 4 threads is only 48% of the single-threaded EMAN. The power consumption data shows that the customized architecture outperforms the CPU and GPU by 7.3 times and 3.4 times, respectively.

Table 6 Power consumption analysis

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

It is worth noting that the GTX480 chip is built with the 40nm process, while the Virtex5 FPGA is built with the less efficient 65nm process. Therefore, the performance and power cost of the FPGA accelerator can be further improved by upgrading to the Virtex6 and Virtex7 families. The area consumption and frequency of the MakeRFP and the RTAlign stream are summarized in Table 7. In practice, the two streams share lots of common blocks and can be combined to form a more compact stream.

Table 7 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 Conclusion

In this paper, a coarse-grained stream architecture is introduced, in which the computing datapath is run-time configurable. In order to decouple the computing workload from the data flow, to a dedicated data flow module (DFM) is resorted to support the functionality of pattern-based data access. Complex functions can be emulated by configuring the datapath of the computing stream multiple times, whereas both data operations and related address calculations are offloaded to the DFMs.

The stream architecture is evaluated by accelerating a large-scale scientific application, which is an open source software package for single-particle 3D reconstruction from Cryo-electron microscopy images, on the customized FPGA accelerator card. The FPGA-based accelerator design is compared with the parallelized versions on a multi-core CPU and an up-to-data GPGPU. Measured in raw performance, the FPGA-based design outperforms a 4-cores CPU by 2.54 times. When compared with the previous GPU-based design, the FPGA-based design is about 3 ~ 4 times slower. However, we argue that it is still beneficial to use the FPGA when taking the 7 ~ 8 times power efficiency into consideration.

References

[1] Li L C, Li X J, Tan G M, et al. 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, San Jose, USA, 2011. 195-204

[2] Scrofano R, Gokhale M, TrouwF, et al. Hardware/Software approach to molecular dynamics on reconfigurable computers. In: Proceedings of the 14th Annual IEEE Symposium, Field-Programmable CustomComputing Ma-

chines, USA, 2006. 23-34
[3] Florent de Dinechin, Cristian K, Bogdan P. Generating high-performance custom floating-point pipelines, In: Proceedings of the Field-Programmable Logic and Applications, Pargue, CzechRepublic, 2009. 59-64
[4] Ludtke S J, Baldwin, P R, et. al. Eman: Semiautomated softwarefor high-resolution single-particle reconstructions. *Journal of Structural Biology*, 1999, 128(1): 82-97
[5] Marathe J, Mueller F. 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, Toronto, Canada, 2008. 1-8
[6] Ryoo S, Rodrigues C, Baghsorkhi S, et al. Optimization principles and application performance evaluation of a multithreaded gpuusing cuda. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, SaltLake City, USA, 2008. 73-82
[7] Tan G M, Sreedhar V C, Gao G. Just-in-time locality and percolation for optimizing irregular applications on a many core architecture. In: Proceedings of the 21st Annual Workshop Languages and Compilersfor Parallel Computing, Edmonton, Canada, 2008. 331-342
[8] Vasily V, James W. Benchmarking gpus to tune dense linear algebra. In: Proceedings of the ACM/IEEE Conference on Super-computing, Austin, USA, 2008. 1-11
[9] Duan B, Wang W, et al. Floating-point mixed-radix FFT core generation for FPGA and comparison with GPU and CPU. In: Proceedings of the 2011 International Conference on Field-Programmable Technology, New Delhi, India, 2011. 1-6
[10] Xilinx, http://www.xilinx.com/
[11] Fluke, http://www.fluke.com/fluke/

Duan Bo, born in 1978, Ph. D candidate, received his B. S. and M. S. degrees from University of Science and Technology of China. His Research directory is reconfigurable computing and computer architecture.