Accelerated Cell Imaging and Classification on FPGAs for Quantitative-phase Asymmetric-detection Time-stretch Optical Microscopy

Junyi Xie[1], Xinyu Niu[2], Andy K. S. Lau[1], Kevin K. Tsia[1], Hayden K. H. So[1]
1 Department of Electrical and Electronics Engineering, University of Hong Kong, Hong Kong
2 Department of Computing, School of Engineering, Imperial College London, UK
Email: xiejunyi@connect.hku.hk, {hso, andyksl,tsia}@hku.hk, niu.xinyu10@imperial.ac.uk

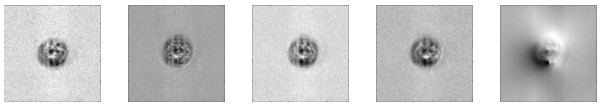
***Abstract*—With the fundamental trade-off between speed and sensitivity, existing quantitative phase imaging (QPI) systems for diagnostics and cell classification are often limited to batch processing only small amount of offline data. While quantitative asymmetric-detection time-stretch optical microscopy (Q-ATOM) offers a unique optical platform for ultrafast and high-sensitivity quantitative phase cellular imaging, performing the computationally demanding backend QPI phase retrieval and image classification in real-time remains a major technical challenge. In this paper, we propose an optimized architecture for QPI on FPGA and compare its performance against CPU and GPU implementations in terms of speed and power efficiency. Results show that our implementation on single FPGA card demonstrates a speedup of 9.4 times over an optimized C implementation running on a 6-core CPU, and 3.47 times over the GPU implementation. It is also 24.19 and 4.88 times more power-efficient than the CPU and GPU implementation respectively. Throughput increase linearly when four FPGA cards are used to further improve the performance. We also demonstrate an increased classification accuracy when phase images instead of single-angle ATOM images are used. Overall, one FPGA card is able to process and categorize 2497 cellular images per second, making it suitable for real-time single-cell analysis applications.**

***Keywords*—quantitative phase imaging, cell classification, time-stretch imaging, image-based single-cell analysis, real-time, ultrafast events, FPGA, GPU, acceleration, power efficiency.**

I. Introduction

In biological imaging applications, quantitative phase imaging (QPI), in contrast to other established bioimaging modalities, offers a promising solution for label-free cell or tissue quantitative assessment with nanometer precision [1]. Fundamentally, QPI operates by measuring the optical phase-shift across the specimen under test. Derived from this phase shift distribution, quantitative information such as cell volume, mass, refractive index, stiffness, and optical scattering properties can subsequently be obtained. Such quantitative information is invaluable as they may serve as a niche set of biomarkers for cellular identification and classification. Unfortunately, with its image acquisition rate intrinsically limited by the speed-sensitivity trade-off in the CCD/CMOS image sensors, current QPI techniques are yet to be fully compatible for high-throughput cellular assays – an unmet need for both basic research and clinical applications where large population of cells must be examined.

978-1-4673-9091-0/15/$31.00



(a) (b) (c) (d) (e)

Fig. 1: (a)–(d): Four single-angle ATOM images of one Chondrocyte with four fiber coupling angles. (e): Retrieved phase image of one Chondrocyte after QPI processing

Recently, a new imaging technique called asymmetric detection time-stretch optical microscopy (ATOM) has been demonstrated for single-cell imaging with an unprecedented frame rate up to MHz [2]. Not only ATOM can bypass the classical speed-sensitivity trade-off in CCD/CMOS image sensors, it can also be extended to perform QPI such that it can provide label-free quantitative information at an ultrafast speed. This new QPI technique, called quantitative-phase ATOM (Q-ATOM), is able to operate at high-speed data rate (>GSa/s) in real-time. To fully exploit the potential of Q-ATOM in real-world scenarios, however, mandates a powerful back-end computing facility that is able to not only acquire such high-volume, high-speed data in real-time, but also to perform complex quantitative phase image retrieval, as well as application specific cell classification on the retrieved data.

To that end, this paper proposes a fully streamable and optimized architecture on FPGA for Q-ATOM processing and cell image classification. To demonstrate the effectiveness of our system, imaging data of three live and label-free (no staining agents for internal details visualization) cells, namely human chondrocytes, human osteoblasts and mouse fibroblasts, were used. The first part of the system produces one phase image of a cell from four single-angle ATOM images (Figure 1). Subsequently, the generated images are processed through a support vector machine (SVM) for classification.

The performance of our implementation was compared against a GPU and a CPU implementation of the same algorithm. With an imaging and classification throughput of 2497 images per second, our implementation on single FPGA card is 9.4 times faster than our optimized C implementation running on a 6-core CPU, and 3.47 times faster than an equivalent GPU implementation. In terms of power-efficiency, the FPGA implementation is 24.19 and 4.88 times more efficient than

the CPU and GPU implementation respectively.

In addition, when compared to a similar imaging system without QPI, the additional QPI processing improves the classification accuracy by $2\,\%$ to $4\,\%$. The improved classification accuracy illustrates the benefits of QPI processing, which motivates this acceleration work.

As such, we consider the main contributions of this work are in the following areas:

- We have design and implemented a novel ultra-fast quantitative phase imaging processing and classification system on FPGAs that is faster and more power efficient than its equivalent implementation on GPU and CPU.
- We have demonstrated the potential of a multi-FPGA implementation of the same algorithm that provides further speedup and is viable for real-time applications.
- We have demonstrated the benefit of QPI processing in terms of increased cell classification accuracy.

II. Background and Related Works

A. QPI Processing Algorithms

Various algorithms for QPI processing has been proposed by researchers [3], [4]. Research [5] has accelerated a QPI algorithm on GPU, and there is no normalization of intensity performed on images. Frequency domain medical imaging algorithm on FPGA is demonstrated in [6]. Algorithms such as [4], [5] and [6] use interferograms which introduce an iterative and computationally intensive phase unwrapping step. QPI in Q-ATOM does not require phase unwrapping and is more suitable for FPGA acceleration. Intensity loss in optical delay lines of Q-ATOM introduce an intensity normalization step which is non-iterative and easy to be implemented on FPGA. In [6], researchers demonstrates advantages of FPGA in processing images in frequency domain. Both [6] and our QPI processing in Q-ATOM involves low pass filtering, forward and inverse FFT. Compared to [6], FFT in our system processes complex numbers as input and consumes more hardware resources.

QPI algorithm proposed in [3] extracts phase images from bright-filed images similar to the ATOM images. Thus we use algorithm in [3] as a guideline to develop our QPI processing algorithm.

B. Linear SVM

Given a set of instances with labels, linear SVM [7] solves the unconstrained optimization problem

$$\min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{l}\xi(\mathbf{w};\mathbf{x}_i;\mathbf{y_i}), \tag{1}$$

where $\xi(\mathbf{w};\mathbf{x}_i;y_i)$ is the loss function and C is the penalty parameter. If the instance-label pairs are linearly separable into two regions on a two dimensional representation, we can maximize the margin between the regions using two hyperplanes. The training phase maximize the margin and generate one hyperplane $\mathbf{w}$. In the testing phase, we classify a instance $\mathbf{x}$ as positive if $\mathbf{w}^T\mathbf{x} > 0$ and negative otherwise [7].

Researchers have used SVM to classify blood cells in bone marrow and demonstrates a high accuracy [8]. Results of [8] inspires us to classify QPI processed phase images of human cells using SVM.

III. Design Approach

A. Top Level Architecture

Our QPI system is composed of a spatial domain module and a frequency domain module as shown in Figure 2. Pixel values of four ATOM images are streamed into FPGA in each clock cycle. In each module, there are several kernels to perform different mathematical operations on images.

We denote the four input single-angle ATOM images as $I\{1,2,3,4\}$ and the output phase image as ϕ. These retrieved phase images are subsequently streamed into an SVM classification kernel to compute a decision value.

Data is streamed into and out from FPGA via PCIe for Xilinx FPGAs and Infiniband for Altera FPGAs.

All computation are performed on FPGA except that SVM models are pre-computed on CPU. We pre-compute a group of models and save them in on-board memory. To classify different types of cells, we simply reload the on-board ROM.

B. Spatial Domain Module

Spatial domain processing module is composed of background subtraction kernel, intensity normalization kernel and complex phase shift extraction kernel.

Firstly, background subtraction in each line is performed by subtracting the mean pixel value of the line from each pixel. Secondly, due to the different intrinsic loss of optical delay lines, the four captured ATOM images are of different average intensity. Therefore, normalization of each individual image is required to eliminate the effects of intensity variation between the four images, which can ultimately affect the phase retrieval accuracy. Intensity normalization of each image is performed by

$$I_{g\_norm} = \frac{I_{bgs} - min(I_{bgs})}{max(I_{bgs}) - min(I_{bgs})} \tag{2}$$

where $min/max(I_{bgs})$ is the minimum, maximum value of the background subtracted image and I_{g_norm} are the intensity normalized images.

Differential phase gradient contrasts images $(I5, I6)$ and absorption contrasts images $(I7, I8)$ can be obtained by subtraction and addition of these normalized images, which are calculated by

$$\begin{aligned} I5 &= I3_{g\_norm} - I4_{g\_norm}, \\ I6 &= I1_{g\_norm} - I2_{g\_norm}, \\ I7 &= I3_{g\_norm} + I4_{g\_norm}, \\ I8 &= I1_{g\_norm} + I2_{g\_norm}. \end{aligned} \tag{3}$$

The wavefront tilt (θ_x, θ_y) and local phase shift ($\nabla\phi_x$, $\nabla\phi_y$) introduced by the flowing cells are calculated based on the

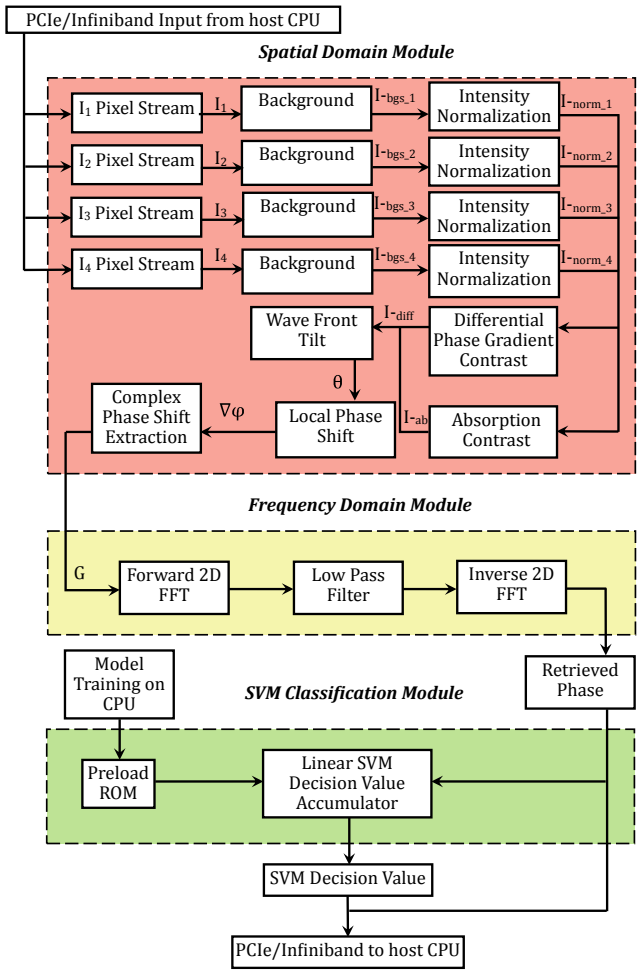


Fig. 2: Top-level Architecture. All matrices are decomposed into data streams. $I_{\{1,2,3,4\}}$ are original single-angle ATOM images. $I_{bgs\_\{1,2,3,4\}}$, $I_{norm\_\{1,2,3,4\}}$, I_{diff}, I_{ab}, θ, $\nabla\phi$, G represent background subtracted images, intensity normalized images, differential phase gradient contrast and absorption contrast, wave front tilt, local phase shift and complex phase shift respectively.

differential phase-gradient contrast and absorption contrast images $I\{5,6,7,8\}$ by

$$\theta_x = \frac{NA}{I7 * (I5 - min(I5))}, \qquad \theta_y = \frac{NA}{I8 * (I6 - min(I6))}, \tag{4}$$

$$\nabla\phi_x = \frac{2\pi}{\overline{\lambda}} * \theta_x, \qquad \nabla\phi_y = \frac{2\pi}{\overline{\lambda}} * \theta_y, \tag{5}$$

where NA is numerical aperture of our system and $\overline{\lambda}$ is the mean illumination wavelength. Lastly, the complex phase shift G in spatial domain is calculated by

$$G = \nabla\phi_x + i * \nabla\phi_y. \tag{6}$$

Hardware implementation of the three kernels are shown in Figure 3.

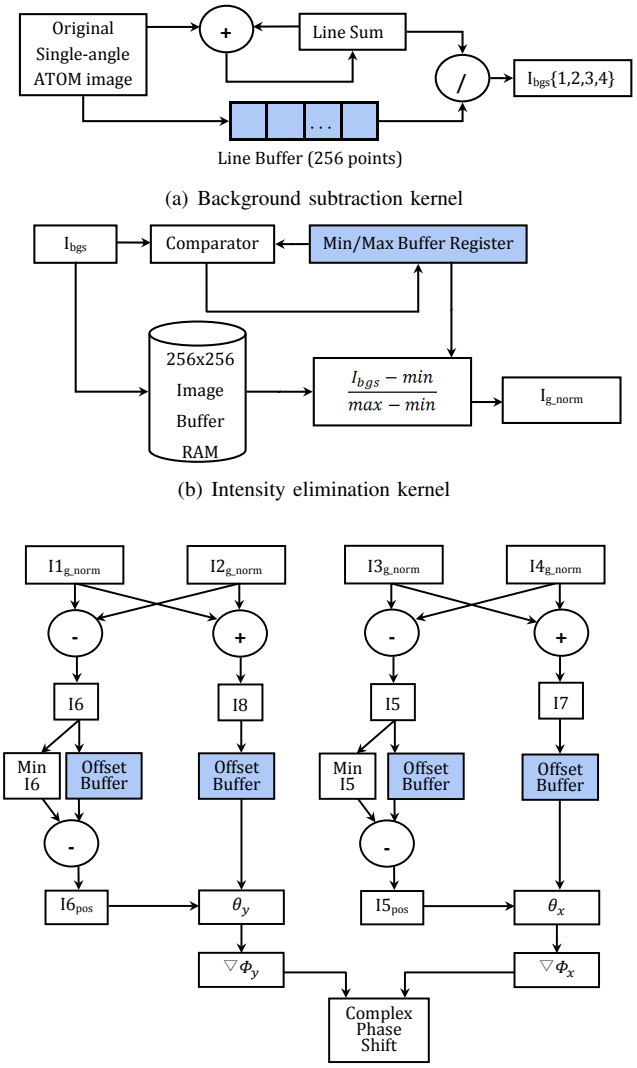


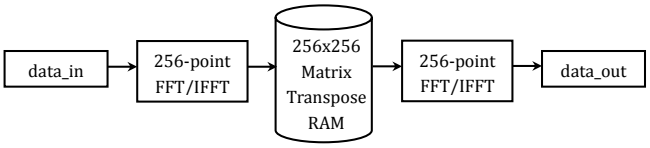
Fig. 3: Architecture of Kernels in Spatial Domain Processing Module

C. Frequency Domain Module

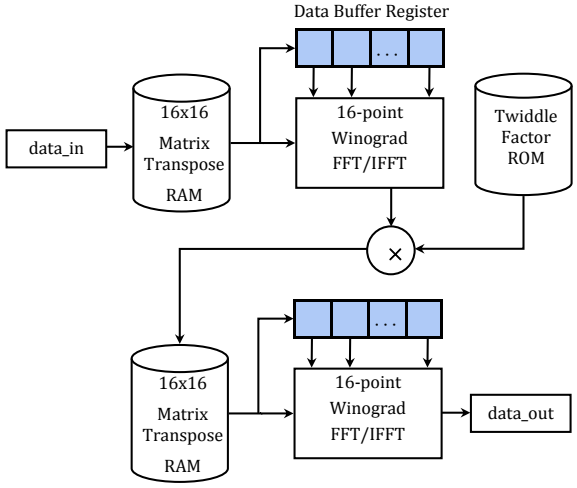
Frequency domain processing involves 2-D forward and inverse FFT and a low pass filter [3] to reduce high frequency noises. According to [3], the final retrieved phase image (Figure 1(e)) $\phi(x,y)$ is given by

$$\phi(x,y) = Im\left[\mathcal{F}^{-1}\left\{\begin{array}{ll} \frac{\mathcal{F}\{G(x,y) * FOV\}}{2\pi(k_x + ix_y)}, & |k \neq 0| \\ C, & |k = 0| \end{array}\right\}\right], \tag{7}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ correspond to forward and inverse FFT and $k_{x,y}$ corresponds to a low pass filter implemented as a linear ramp. FOV is the field of view of the imaging system and the value is 80 microns. Forward/inverse 2-D FFT of 256×256 points (Figure 4) consists of two 256-point 1-D FFT/IFFT. Each radix-16 256-point FFT consists of two 16-point FFT



(a) 2-D FFT kernel



(b) Radix-16 256-point FFT kernel

Fig. 4: Architecture of 2-D Fast Fourier Transform

and is expressed by

$$X(k) = X(16r+s) = \sum_{m=0}^{15} W_{16}^{mr}\ W_{256}^{ms} \sum_{l=0}^{15} x(16l+m) W_{sl}^{16}, \quad r = 0\ to\ 15, s = 0\ to\ 15, \tag{8}$$

where W is the complex twiddle factor calculated by

$$W_{256}^{ms} = cos(\frac{2\pi ms}{256}) \pm jsin(\frac{2\pi ms}{256}), \quad m, s = 0, 1, 2, 3, ..., 255. \tag{9}$$

D. Memory Management

All the images we use in the system are of 256×256 size. A large amount of temporary data will be generated (Table I) in order to unwrap loops and solve data dependencies. In background subtraction kernel, one column of the image matrix needs to be buffered and the subtraction can be performed after the mean value is calculated. In intensity normalization kernel, the whole matrix needs to be buffered because the maximum and minimum pixel value are obtained after the whole matrix is streamed to FPGA. In complex phase shift extraction kernel, differential phase gradient contrast I_5 and I_6 are obtained by subtraction which contains negative values. We need to subtract the minimum value from the contrasts to make every value positive before calculating wave front tilt $\theta_{x,y}$ and local phase shift $\nabla\phi_{x,y}$. Again, the whole matrices of I_5 and I_6 need to be buffered.

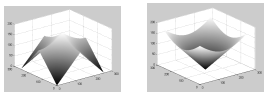
2-D FFT in frequency domain processing module involves a row major transformation followed by a column major transformation. In a fully streamable implementation, we have only one complex number streamed into FFT kernel in each clock cycle. Thus, two matrix buffers are needed to perform matrix transpose in FFT. When the previous matrix is read in column major, current matrix is written into memory in row major.

Each SVM model have the same size as the retrieved phase image. Number of models to be stored in on-board ROM depends on user specification. Moreover, we have to shift the four quadrants of the frequency spectrum (FFT shift) so the low frequency components are at the center of spectrum before filtering and at the four corners after filtering. Rearranging of 2-D spectrum in a data stream costs a considerable amount of on-board RAM to store the temporary spectrum. To save memory usage, we rearrange the low pass filter and apply the new filter directly on the data stream of original spectrum using a counter chain (Figure 5).

TABLE I: Temporary data in each kernel

Kernel	Data Size[1]	Memory Type[2]
Offset Data Stream	vary in kernels	Register
Background Subtraction	256x4 real	Register
Intensity Normalization	2x256x256x4 real	BRAM
Matrix transpose before FFT	2x256x256 complex	BRAM
Matrix transpose after FFT	2x256x256 complex	BRAM
FFT Shift before LPF[3]	2x256x256 complex	BRAM
FFT Shift after LPF[3]	2x256x256 complex	BRAM
SVM	256x256 real	BRAM

1 real and complex represent real and complex number. 2 SVM model are preloaded into read-only ROM before each run. Both ROM and RAM are implemented using on-board BRAM. 3 Memory consumption of FFT shift is eliminated.



(a) (b)

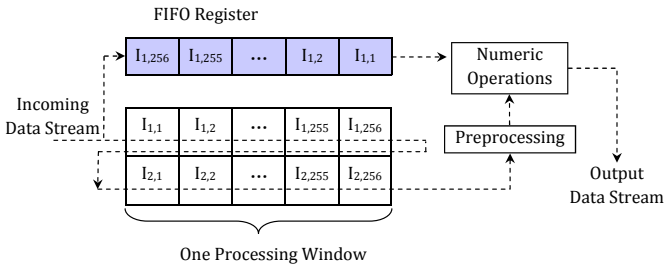
Fig. 5: (a) original LPF: low frequency components at center of spectrum, (b) rearranged LPF: low frequency components at four corners of spectrum

In order to handle the large amount of temporary data, we introduce a mixed memory configuration composed of registers and block RAMs (BRAM) (Figure 6). In each kernel, matrices are saved in on-board BRAM pixel by pixel in row major and offset data stream are implemented using buffer register. In order to reduce memory consumption, we convert the data in 45-bit fixed point number to 32-bit floating point number before writing data to BRAM and convert in reverse after reading data from BRAM.

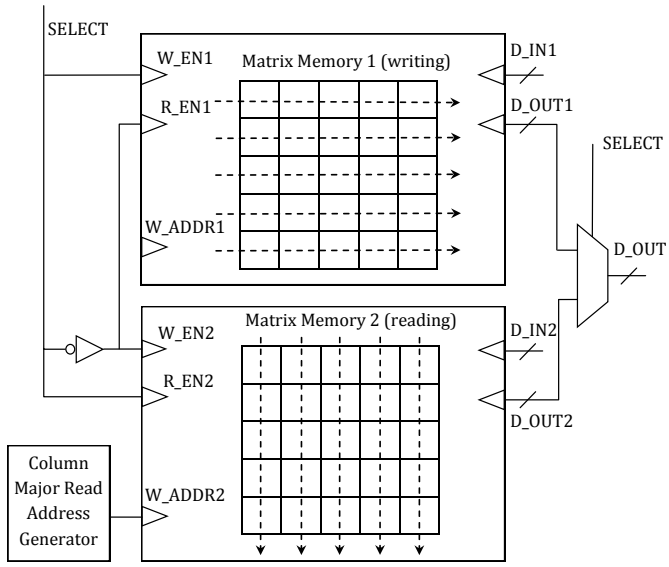
IV. Design Optimization

A. Number Representation Scheme

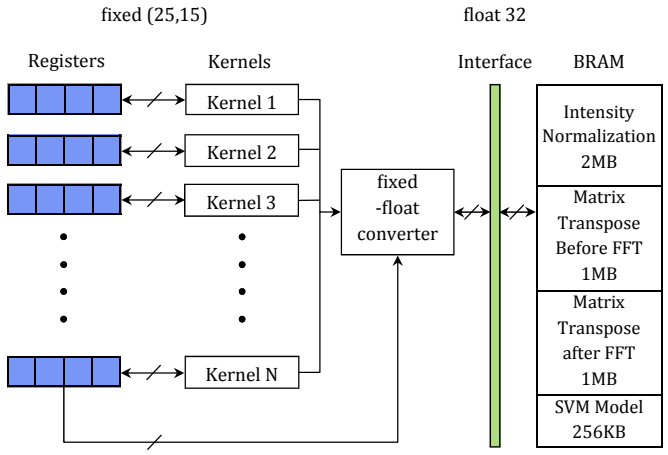
Floating point number representation is used for verification of QPI algorithm on Matlab. However, floating point number arithmetic consumes much more hardware resources than fixed point arithmetic on FPGA. The whole system cannot be placed on FPGA using full floating point number representation. Dynamic range of all the numeric operations are estimated on CPU. The largest number appears in the frequency spectrum



(a) Usage of FIFO registers to solve data dependency



(b) Matrix transpose memory in 2-D FFT kernel. While we write data into one memory block, another memory block is read in a transposed sequence.



(c) Overall memory configuration. Each kernel has its own memory interface to access BRAM.

Fig. 6: Memory configuration

after 2-D FFT at 10^6 scale, which requires 23 integer bits. We run simulations using different number of fractional bits. Simulations show the accuracy of the final retrieved phase image are not influenced when more than 12 fractional bits are used. Thus, the dynamic range requires a (23, 12) fixed point number representation for sufficient accuracy. For a higher accuracy redundancy, we select a (25, 15) fixed point number representation.

TABLE II: Design model parameters

model parameters	
N_{dp}	number of replicated kernels/datapaths
i, f	number of exponent (integer) and mantissa (fractional) bits
N_{fpga}	number of used FPGAs
design properties	
$Usage$	hardware resource usage
BW	bandwidth requirements
$latency$	computation delay/latency
TH	computation throughput
design constants	
U_r	usage of resource type r. $r \in$ FFs/LUTs/BRAMs/DSPs
$op_{\oplus,ker}$	number of operation $\oplus \in \{+ - \times \div\}$ in kernel ker
$kernels$	number of kernels in a data-path
A_r	available resources of type r
$freq$	operating frequency
N_{ops}	number of arithmetic operations

B. Design Models

In order to optimize designs under resources and bandwidth constraints, we develop design models to estimate the throughput, resource usage, and data bandwidth requirements with design parameters. Table II lists the design parameters and estimated design properties. In the hardware design, we pipeline the design kernels to ensure that after an initial delay, one application data-path (i.e. pipelined design kernels) processes one image pixel set per clock cycle. In our application, each image pixel set contains 4 images pixels, one for each single-angle image. Given an optimized design with N_{dp} data-paths in each FPGA and N_{fpga} FPGAs, throughput TH can be expressed as:

$$TH = freq \cdot N_{fpga} \cdot N_{dp} \cdot N_{ops} \tag{10}$$

where $freq$ indicates the design operating frequency, N_{ops} indicates the numbers of arithmetic operations to process one image pixel set.

In practice, bounded by the available resources and bandwidth, the theoretical throughput described in Eq. 10 cannot always be achieved. In order to properly optimize the hardware design, we further develop design models for resource usage and bandwidth requirements. For a hardware design using representation format t with i integer bits and f fractional bits, the resource usage for arithmetic operations can be estimated by accumulating the resources used by each arithmetic operator,

$$U_{LUTs/FFs/DSPs} = \sum_{ker=1}^{kernels} \sum_{op \in \oplus}^{op_{\oplus,ker}} U_{LUTs/FFs/DSPs} + I_{LUTs/FFs/DSPs} \tag{11}$$

where $I_{LUTs,FFs,DSPs}$ indicates the resource usage of communication infrastructures, such as PCI-e drivers and off-chip memory controllers.

The on-chip memory usage is determined by the kernel data dependencies. As shown in Table I, the dependent data are

buffered as data stream in. This consumes on-chip memory resources, and introduces an initial delay to fill in all the data buffers. In order to save memory usage, we convert fixed point number of 40 bits to 32-bit floating point number before writing to memory. The memory usage can be estimated with the kernel buffer size, and expressed as:

$$U_{BRAM} = \sum_{ker=1}^{kernels} dep_{ker} \cdot R/C \cdot (i+f) + I_{BRAM} \quad (12)$$

where dep_{ker} indicates the dependent data size as list in Table I, and $(i+f)$ indicates the number of bits for each data item. $Real/Complex$ is 1 for real numbers, 2 for complex numbers.

In correspondence to the on-chip buffer size, the initial computation delay can be estimated as the time to fill the buffers:

$$latency = \sum_{ker=1}^{kernels} dep_{ker} \quad (13)$$

After an initial delay, at each clock cycle, each data-path streams in and processes one image pixel set per clock cycle. Correspondingly, the data bandwidth requirements can be expressed as:

$$BW = freq \cdot N_{dp} \cdot N_{image\_stream} \cdot N_{fpga} \quad (14)$$

When the optimized design uses distribute workload across multiple FPGAs, each FPGA processes separate image stream independently. BW indicates the overall bandwidth requirements of our target FPGA system.

C. Multiple FPGAs

We further test the scalability and improve the performance of our QPI and cell classification system by processing images in parallel on multiple connected FPGAs. Each computing node has 8 FPGA cards and we use separate channels to stream data into each FPGA card. When four or less FPGA cards are used, processing throughput increases linearly. When more than four FPGA cards are used, throughput stops increasing because bandwidth limit is reached. To find the optimal number of FPGA cards for efficient parallelization, a model is developed. Bounded by available resources $A_{LUT/FF/DSP}$, memory M_A and communication bandwidth BW, the model is expressed by

$$\begin{aligned} &\textbf{maximize: } \frac{freq \cdot N_{fpga}}{image\_size} \\ &\textbf{subject to:} \\ &U_{ops} \cdot N_{ops} \cdot N_{dp} + I_{LUT/FF/DSP} \leq A_{LUT/FF/DSP} \\ &U_{BRAM\_bgs\_sub} + U_{BRAM\_norm} + U_{BRAM\_FFT\_trans} \cdot 2 \\ &\qquad + U_{BRAM\_SVM} + I_{BRAM} \leq A_{BRAM} \\ &N_{fpga} \cdot N_{pipe} \cdot max(4 \cdot IO_{in\_pixel}, IO_{out\_pixel}) \cdot freq \\ &\qquad \leq BW \end{aligned} \quad (15)$$

D. Choice of FFT Algorithm

The 2-D 256×256 point FFT is composed of two 256-point FFT. Each 256-point FFT is composed of two 16-point FFT which is the basic building block of the whole FFT kernel. In total, four forward 16-point FFT kernels and four inverse FFT (IFFT) kernels are needed. The commonly used radix-2 and radix-4 16-point FFT process data in parallel which consumes a large amount of hardware resources to implement 16 complex multipliers and data path. To reduce resource usage, we use 16-point Winograd FFT [9] which requires only 10 complex multiplications and 74 additions. With a pipelined design, usage of complex multipliers can be further reduced. Resource usage of one 16-point Winograd FFT (IFFT) on Altera Stratix V GS 5SGSD8 FPGA in Max4 station is shown in Table III.

TABLE III: Resource usage of one 16-point Winograd FFT kernel

	LUTs	FFs	BRAMs	DSPs
Resources available	524800	1049600	2567	1963
Resources used	9324	14843	9	117
Percentage	1.78%	1.41%	0.35%	5.96%

V. Hardware Implementation Results

A. Testing Platforms

TABLE IV: Testing platform details

Platform	CPU[1]	RAM	Compiler	APIs	batch size[2]
CPU 1-thread	Intel Xeon E5-2640 (6 cores, 15 MB cache)	64GB	GCC	FFTW (1-thread)	8000x4
CPU 6-thread	Intel Xeon E5-2640 (6 cores, 15 MB cache)	64GB	GCC	FFTW (6-thread)	8000x4
GPU (Nvidia Tesla K40C)	Intel Xeon E5-2650 (8 cores, 20 MB cache)	64GB	GCC, NVCC[3]	cuFFT, cuBLAS	3000x4
FPGA (Altera Stratix V in Maxeler Max4 Station)	Intel Xeon E5-2640 (6 cores, 15 MB cache)	64GB	GCC, Maxcompiler[4]	Maxcom-piler	8000x4

1 For FPGA and GPU implementation, CPU performs as host processor. 2 Number of single-angle ATOM images processed in each run. 3 NVCC: Nvidia CUDA Compiler. 4 Maxcompiler: Compiler developed by Maxeler Technologies to compile high level synthesis code.

QPI processing and classification algorithm is implemented on CPU, GPU and FPGA using single-thread C code, 6-thread optimized C code, Nvidia CUDA C code and Maxeler maxj high level synthesis code. Details of each testing platform are listed in Table IV. A Nvidia Tesla K40C specialised computing GPU is used as GPU platform. Both CPU and GPU implementation use 32-bit single-precision floating point number representation.

To optimize CPU and GPU implementation, multi-thread processing on CPU is implemented using OpenMP [10] multi-processing application program interface (API) developed by Intel. $O3$ flag of GCC compiler is turned on for maximum optimization. FFT is implemented in C code on CPU using FFTW library [11] which is a standard high performance FFT library and on GPU using cuFFT [12] which is a GPU accelerated implementation of FFT. FFTW also supports multi-thread processing using OpenMP and we turned the OpenMP

flag on when compiling FFTW library. Matrix algebra in GPU code is optimized using CUDA Basic Linear Algebra Subroutines (cuBLAS). Various optimization techniques are used in GPU implementation such as reduction to increase GPU performance.

Hyper-threading is a technology on Intel CPU to virtualize two logical cores on one physical core. We measure throughput of CPU when hyper-threading is turned off to prevent thread migration between physical cores.

Raw single-angle ATOM images are streamed into FPGA and GPU in batches. Batch sizes are different on FPGA-based and GPU-based implementation due to different memory sizes of the devices. Host memory of FPGA is 64 GB. Nvidia Tesla K40C GPU has 12 GB high speed double data rate type five synchronous graphics memory (GDDR5).

B. Resource Usage on FPGA

In Table V, we list resource usage of each fixed point operation $\oplus \in \{+ - \times \div\}$, estimated resource usage using Eq. 11, Eq. 12 and measured resource usage.

TABLE V: Resource usage on single FPGA

	LUTs	FFs	BRAMs	DSPs
Resource usage of +	59	59	0	0
Resource usage of -	17	17	0	0
Resource usage of $\times$	68	150	0	4
Resource usage of $\div$	2025	3443	9	0
Estimated Resource usage	93118	192367	1921	944
Measured resource usage	116242	232120	2063	1041
Resources available	524800	1049600	2567	1963
Percentage Usage	22.15%	22.12%	80.37%	53.03%

C. Throughput, Bandwidth and Power Consumption

We measure speed of our QPI processing and classification system in terms of throughput. Throughput is defined as the number of QPI phase images that a system generates and classifies per second (pcs per sec) and floating point operations per second (FLOPS). Using Eq. 10, optimal throughput of QPI system on single FPGA can be calculated by

$$\begin{aligned} TH &= 1.88 \times 10^6 \times 1 \times 1 \times N_{ops} \\ &= 36.85(GFLOPS) = 2868.7(pcs\ per\ sec) \end{aligned} \quad (16)$$

Multiple FPGA cards share a common Infiniband switch to connect to CPU and we do not use the communication channel between FPGA cards. Bandwidth consumption is measured by calculating the product of bandwidth consumption of retrieving one phase image and measured throughput in pcs/sec. As shown in Figure 7, throughput stops increasing linearly when more than three FPGA cards are used, and communication bandwidth becomes the major constraint. Detailed Infiniband bandwidth consumption is shown in Table VI. In Figure 7,

TABLE VI: Actual host/FPGA Infiniband bandwidth consumption

# of FPGA card	1	2	3	4	5	6	7	8
Avg. BW (MB/s)	1569	2808	4064	5248	5159	6005	5773	6140

it shows throughput is bounded by available bandwidth. If bandwidth is infinite and all latency is eliminated, theoretical

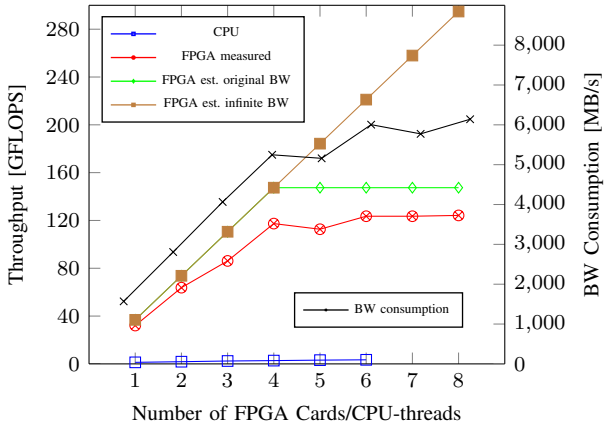


Fig. 7: Throughput of CPU and Multiple FPGA Cards

peak performance of 8 FPGA card can be further improved to 294.8 GFLOPS.

Total power consumption was measured on a node of 8 FPGA cards. We obtained dynamic power of one FPGA board by subtracting the static power of the unused Maxeler cards (the average static power per card is 19.6 W).

Performance summary is shown in Table VII. Implementation on single FPGA card demonstrates a 9.4 times speedup compared to 6-thread CPU implementation and 3.47 times speed up compared to GPU implementation in terms of throughput. FPGA implementation achieve total power efficiency (FLOPS per Watt) of 24.19 times higher than 6-thread CPU implementation and 4.88 times higher than GPU implementation. Dynamic power efficiency of FPGA implementation is 16.81 times and 5.17 times higher than 6-thread CPU and GPU implementations respectively (Table VII).

D. Precision and Image Quality

Retrieved phase images should have no difference on CPU and GPU implementations. FPGA implementation converts the phase image from fixed point number to floating point number before streaming back to host. Simulation and actual hardware implementation on FPGAs show the phase image has no major difference compared to images retrieved by base-line CPU implementation.

E. Classification Accuracy

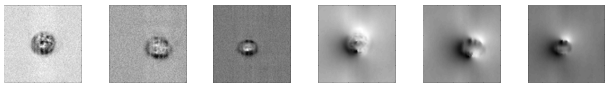
Three types of unstained, live-cells images are used to validate the accuracy of classification, namely human chondrocytes (OAC), human osteoblasts (OST) and mouse fibroblasts (3T3) from cell-line (Figure 8). All three types of cells look similar and are hardly classifiable with human eyes. We process 1500 ATOM images for each type of cell to retrieve the phase images and use single-angle ATOM images and retrieved phase images of each type of cell to train SVM models. Accuracy of classifying any two types are measured using either single-angle ATOM image model or retrieved phase image model. Cross validation of 10 folds (nine folds for model training and one fold for model testing) are conducted using linear SVM to get an average classification

TABLE VII: Performance Summary

	CPU (1-thread)	CPU (6-thread)	GPU	1 FPGA	4 FPGA	8 FPGA[1]
clock frequency (GHz)	2.5	2.5	0.745	0.188	0.188	0.188
throughput (pcs/sec)[2]	100.62	265.03	719.21	2497.38	9136.83	22949.74
throughput (GFLOPS)[3,4]	1.29	3.41	9.24	32.08	117.37	294.80
speedup	1×	2.64×	7.15×	**24.82**×	**90.98**×	**228.08**×
total power (W)[5]	411	457	250	177.8	227.8	294.46
total power efficiency (MFLOPS/W)	3.14	7.46	36.96	180.43	515.23	1001.15
total power efficiency improvement	1×	2.38×	11.77×	**57.46**×	**164.09**×	**318.84**×
dynamic power (W)	38	84	70	47	97	163.67
dynamic power efficiency (MFLOPS/W)	33.95	40.60	132	682.55	1210.00	1801.19
dynamic power efficiency improvement	1×	1.20×	3.89×	**20.10**×	**35.64**×	**53.05**×

1 We assume theoretical peak performance and power consumption increase linearly if bandwidth is infinite and all latency is eliminated.
2 Number of phase images (256×256) retrieved per second.
3 Time of data transfer and device configuration is included to calculate throughput.
4 Since FPGAs use fixed-point number with same accuracy as floating point number, we use FLOPS as unit of FPGAs' throughput.
5 Total power consumption includes both static power (130.8 W) and dynamic power.

accuracy. Classification accuracy of OAC vs. OST, OAC vs. 3T3 and OST vs. 3T3 increased by $3.91\,\%$, $3.47\,\%$ and $2.19\,\%$ respectively when we use models trained with retrieved phase images (Table VIII).



(a) (b) (c) (d) (e) (f)

Fig. 8: (a-c) single-angle ATOM image of OAC, OST, 3T3, (d-f) retrieved phase image of OAC, OST and 3T3

TABLE VIII: Classification Accuracy

Cell Types	linear SVM accuracy
OAC vs. OST (single-angle ATOM image)	86.10%
OAC vs. OST (retrieved phase image)	90.01%
OAC vs. 3T3 (single-angle ATOM image)	91.20%
OAC vs. 3T3 (retrieved phase image)	94.67%
OST vs. 3T3 (single-angle ATOM image)	89.65%
OST vs. 3T3 (retrieved phase image)	91.84%

VI. Conclusion and Future Works

Our novel QPI and SVM classification system on FPGA demonstrates significant speedup and improvement of power efficiency against CPU and GPU. Classification accuracy using QPI processed phase images demonstrates an increase from using original single-angle ATOM image with classifiers. Architecture design and optimization techniques in our system can be adapted to other image processing systems on FPGA. In the future, we will use off-chip DRAM to buffer data to support more pipelines. Convolutional neural network can be implemented on FPGAs as classifier.

Acknowledgement

This work was supported in part by the Research Grants Council of Hong Kong, project ECS 720012E, and by the Croucher Innovation Award 2013. The support of EPSRC grant EP/I012036/1, EP/K011715/1, and the European Union Horizon 2020 Programme under Grant Agreement Number 671653 is gratefully acknowledged.

References

[1] G. Popescu, "Quantitative phase imaging of nanoscale cell structure and dynamics." *Methods Cell Biol*, vol. 90, pp. 87–115, 2008.

[2] T. T. W. Wong, A. K. S. Lau, K. K. Y. Ho, M. Y. H. Tang, J. D. F. Robles, X. Wei, A. C. S. Chan, A. H. L. Tang, E. Y. Lam, K. K. Y. Wong, G. C. F. Chan, H. C. Shum, and K. K. Tsia, "Asymmetric-detection time-stretch optical microscopy (ATOM) for ultrafast high-contrast cellular imaging in flow," *Sci. Rep.*, vol. 4, 01 2014. [Online]. Available: http://dx.doi.org/10.1038/srep03656

[3] A. B. Parthasarathy, K. K. Chu, T. N. Ford, and J. Mertz, "Quantitative phase imaging using a partitioned detection aperture," *Opt Lett*, vol. 37, no. 19, pp. 4062–4064, Oct 2012.

[4] S. K. Debnath and Y. Park, "Real-time quantitative phase imaging with a spatial phase-shifting algorithm," *Opt. Lett.*, vol. 36, no. 23, pp. 4677–4679, Dec 2011. [Online]. Available: http://ol.osa.org/abstract.cfm?URI=ol-36-23-4677

[5] H. Pham, H. Ding, N. Sobh, M. Do, S. Patel, and G. Popescu, "Off-axis quantitative phase imaging processing using CUDA: toward real-time applications," *Biomed. Opt. Express*, vol. 2, no. 7, pp. 1781–1793, Jul 2011. [Online]. Available: http://www.osapublishing.org/boe/abstract.cfm?URI=boe-2-7-1781

[6] A. E. Desjardins, B. J. Vakoc, M. J. Suter, S.-H. Yun, G. J. Tearney, and B. E. Bouma, "Real-time FPGA processing for high-speed optical frequency domain imaging," *IEEE Trans Med Imaging*, vol. 28, no. 9, pp. 1468–1472, Sep 2009.

[7] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, Jun. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1390681.1442794

[8] S. Osowski, R. Siroic, T. Markiewicz, and K. Siwek, "Application of support vector machine and genetic algorithm for improved blood cell recognition," *Instrumentation and Measurement, IEEE Transactions on*, vol. 58, no. 7, pp. 2159–2168, July 2009.

[9] S. Winograd, "On computing the discrete fourier transform," *Mathematics of computation*, vol. 32, no. 141, pp. 175–199, 1978.

[10] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, Jan 1998.

[11] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, Feb 2005.

[12] [Online]. Available: https://developer.nvidia.com/cuFFT