Object Detection using SURF Features on Intel Xeon Phi

By
Rishab Goel, Amol Gupta and Rupa Malladi
Electrical and Computer Engineering Department, University of Florida

Abstract

Object detection and feature matching is a crucial and versatile application in the image processing and recognition domain. The Speed up Robust features (SURF) kernel is fast and popular feature detector and extractor used for object detection. In the High Performance Computing domain, Intel Xeon Phi was released in 2012 as a new competitor to GPUs. It has given performance close to GPUs or even better in many benchmarks and in computation intensive applications. It has additional advantage of offering multiple programming models for the developers to program like OpenCL, OpenMP (Offload & Native), MPI and Cilk threads, whichever suits the developer the most. The aim of this project is to evaluate the programming models on Intel Xeon Phi for SURF algorithm and implement a feature matching application on Intel Xeon Phi. The programming models we analyze are OpenCL (Offload), OpenMP (Offload) and OpenMP (native). We would be comparing the constraints, flexibility and performance of these programming models for SURF algorithm.

Introduction

Object detection is a problem that finds its applications in medical diagnosis and imaging, astronomical data analysis, as well as forensic and criminal investigations. Feature extraction and matching is a widely studied and used technique for object detection. Intel Xeon Phi is a new device and the architecture which is gaining popularity in the supercomputing domain as an alternative to GPUs. Intel Xeon Phi Coprocessor (MIC) [1] is a device architecture which is creating a lot of buzz due to its speed and scalability for numerical computations as well as for recycling of the existing heritage code. Intel Xeon Phi Coprocessor is pitted against NVIDIA GPUs in the high performance computing domain for performance and usage.

We have implemented Large Scale Object Detection using Speeded-Up Robust Features or SURF [2] algorithm with Feature Matching on Intel Xeon Phi Coprocessor (MIC) architecture. We have compared our implementation of OpenMP (Offload & Native) and OpenCL for Intel Xeon phi in terms of speed and constraints. Offload models have Intel Xeon E5 as host while native model have Intel Xeon Phi itself as host. The communication of data for offload model happens using the PCI express.

There have been many varied and fast implementations on GPUs for the image/video classification, but exploring Intel Xeon Phi for this purpose was a new challenge. We explored the native and offload programming model for Intel Xeon Phi coprocessor for the application of object detection. We have exploited the advantages of these programming models and have also looked into the limitations of these programming languages and the model.

We have extended object detection for the purpose of object classification and recognition for large amounts of data on the Intel Xeon Phi coprocessor. The range of programming languages and libraries offered on this platform made it interesting to explore that which model could extract most computing power. Moreover, it is paired with an Intel Xeon server host which is in itself a highly parallelized and multi-threaded device already having great popularity, thus enhancing its usability and attractiveness. The object detection is not only data intensive by itself, but also forms the core of many other such data intensive applications. So, it became an interesting prospect to explore a new high performance computing device architecture for object detection.

Background on the problem

Intel Xeon PhiTM Coprocessor

Intel Xeon Phi is a Many Integrated Core Architecture (MIC). Intel Xeon Phi 5110p (Many Integrated Cores (MIC) architecture) coprocessor contains 60 cores on a chip with capability of scheduling 4 threads per core. It hit 1.01 TFLOPS for double precision floating point with a memory bandwidth of 320 GB/s at 225W. The cores are in-order dual issue x86 architecture with 64 bit support are coupled by On-Die bidirectional Interconnect ring buffers to make x86 SMP-on-Chip.

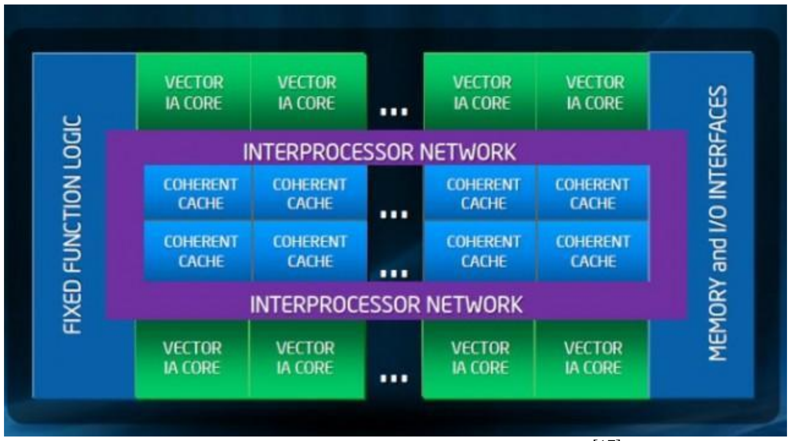


Fig1. Intel Xeon Phi MIC Architecture [17]

The native execution model is defined as the model in which entire application is running on the Xeon Phi coprocessor and there is no communication between coprocessor and CPU. An application can be modified to run on this coprocessor with minimal changes. Availability of large number of cores is beneficial for an application but the program should fit in the limited memory available on the processor. Offload model or Language extension for offload is the model where the application is running on the host processor and the programmer designate the sections of code that need to run on these MIC coprocessors. In this model, there is no sharing of memory in between the CPU and MIC coprocessor and is ideal for large applications containing serial processing. All this parallelism must be provided at the compile time.

Following parallel programming models are running on Intel Xeon Phi MIC and are the focus of our project:

OpenMP [3]*:* OpenMP model is implemented as a set of directives which are recognized by compilers and interpreted to directions, in order to create parallelized tasks which results in speedier execution of a program. TBB or Thread Building Blocks specify parallelism at task level, but on the same processor.

OpenCL [11]*:* OpenCL is the programming language framework that is designed to create programs for heterogeneous parallel computing systems and its strength is its ability to provide scalability to program created for microcontrollers to be scaled to CPUs and GPUs and further even to GPGPUs without much modification to code.

Intel Xeon Phi Optimization Areas

a) *Compiler Optimization:* Scalar optimization includes improving the performance of a single processor using compiler directives/flags during compilation or before runtime. The scalar optimizations include prefetch level configuration, thread affinity [8], pointer aliasing and number of threads to be scheduled.

b) *Vectorization* [10]*:* Explicit vectorization is done using intrinsics but it is suggested to code keeping SIMD lanes of 512 bit SIMD lanes in mind. The Intel compiler inherently attempts to auto-vectorize the code written by the developer. OpenCL and OpenMP programming models both allow auto vectorization.

c) *Multithreading:* INTEL Xeon a single core is programmed such that the tasks could be scheduled on its 60 cores each supporting 4 in-order thread parallelism which amounts to 240 thread availability for the system. Multi-threading introduces one of the problem of False Sharing when more than one threads access the same cache line and cause coherent issues. This problem can be resolved using padding and by keeping the variables private.

d) *Memory Access:* If the data is provided by the DDRRAM (main memory) instead of the cache, then despite code being vectorized, it will face bottleneck at the memory access stage. The efficiency of a code also increases when the data is accessed in a serial order from the memory. The data structures are changed from array of structures to structures of arrays to allow for such serial order. Prefetching of the data and data reusing techniques should be employed to reduce memory access.

e) *Communication:* There are two ways to program the Intel Xeon Phi coprocessors in native model i.e. host and accelerator and the offload model i.e. as an accelerator. The communication during the offload programming the code should be minimized by data reusing and asynchronous memory chunk transfers.

Speed Up Robust Features or SURF

Feature extraction is a major step for image classification as it contains the necessary information regarding the shape of the object and it makes it easy to process the images in a procedure and basically reduces the dimensionality of the image but contains all the relevant features. [2, 15]. The feature descriptor stage is based upon the SIFT descriptor and has been tuned for high recognition rates, whereas the feature localization stage uses the Hessian approximation to be tuned towards parallelism and performance.

For feature extraction we first need to generate feature descriptor vectors using SURF. The SURF method of extracting the relevant feature points consists of multiple stages. The first step is building an integral image with very less memory accesses for the fast box filtering. IΣ(x) at a location x = (x, y) > is a replication of the sum total of all the pixels in the input image computed within a rectangular region bounded by origin and x.

$$I\Sigma(x, y) = \sum_{i=0}^{x} \sum_{j=0}^{y} I(i, j)$$

Fig 2: Formula for calculating integral Image [20]

In order to parallelize the integral image algorithm, SURF is doing calculation in multiple steps in the up sweep and down sweep phases, first along the rows and then along columns, by increasing d from 0 to logn-1 and then decrease from logn-1 to 0. In every step different threads access different array items to perform the calculations concurrently and synchronizations between each step ensures veracity of data. Three additions after the image integral is calculated, the sum over the rectangular area is calculated. Using the box filters.

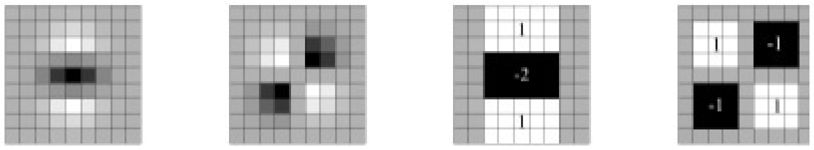


Fig 3: The figure shows the Gaussian second order partial derivatives in y and xy direction and the approximations for the second order partial derivatives in y and xy direction. In the above figure the grey regions represent zero [2].

The next step is to search for distinct feature points by using a SURF detector. For interest point detection the SURF algorithm uses Hessian matrix operation. Hessian Matrix based Input Points: The Hessian Matrix H(x, σ) in x at scale σ for a point (x, y) defined in an image I, where H(x, σ) is given by

$$\mathcal{H}(x,y) = \det \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

Fig 4: Hessian Matrix Representation [20]

$$\mathcal{H}(x) = \mathcal{H} + \frac{\partial \mathcal{H}^T}{\partial x} \boldsymbol{x} + \frac{1}{2} \boldsymbol{x}^T \frac{\partial^2 \mathcal{H}}{\partial x^2} \boldsymbol{x}$$

$$\hat{\boldsymbol{x}} = \frac{\partial^2 \mathcal{H}^{-1}}{\partial x^2} \frac{\partial \mathcal{H}}{\partial x}$$

Fig 5: Equation: Parameterization of Gaussian Kernels [20]

Where Lyy(x, σ) is the replication of convolution of the Gaussian second order derivative (∂2/∂x2)(g(σ))

SURF [9] based implementation suggests that values of the Hessian matrix are pixel independent, thus can be computed in parallel. As the size of the filter increases the number of values that need to be computed gets smaller. For constructing the scale space program intensively accesses the global memory and one of the major factor which affects the bandwidth is whether or not we are accessing the global memory in a coalesced mode. Global memory is divide into segments of 32, 64 and 128 bytes generally. All threads must access same segment of memory and only then we will be able to take advantage of bandwidth. Otherwise memory is accessed in a non-coalesced mode which hurts bandwidth.

To locate the significant points, SURF uses the determinants from the matrix equation in Fig5. The kernel function is parallelized by keeping the thread number equal to the number of pixels that are to be detected. The threads process each value separately as each octave contains different number of values. As soon as the feature point is found the atomic value is increased to get unique index which would be used to store the feature vectors in a global array. For parallelism in calculation of the feature descriptors, two different approaches are used in SURF paper[9]. The first approach is rewriting the innermost loop as a kernel. The second approach is by dividing the original program into three kernel functions. As per [9] each feature point could be assigned to each thread to calculate its orientation in parallel.

Feature Matching

Feature matching is a crucial part of object and feature matching algorithm and have multiple implementations for the same. One is brute force feature matching using least Euclidean distance between the feature points and was optimized later using K-Nearest Neighbor algorithm.

Approach

Speed Up robust Features have multiple implementation in NVidia and AMD GPUs in CUDA and OpenCL implementations. Though the challenges for a GPU implementation are different as compared to Intel Xeon Phi. A typical GPU have more than 2000 of light weight SMP cores, whereas Intel Xeon phi having 60 cores with 512 bit SIMD registers. It is to be noted that Intel Xeon Phi's each core is more powerful than a single SMP cores of NVidia. The biggest issues with Intel Xeon phi is the memory access due to high main memory latency and Ring Interconnect which introduces a bottleneck in its total memory access bandwidth. We come across some bandwidth bound kernels like STREAM TRiad benchmark or Histogram or Prefix Sum[25(Colfax) with O(N) complexity, though having still have aligned access and little but similar computations.

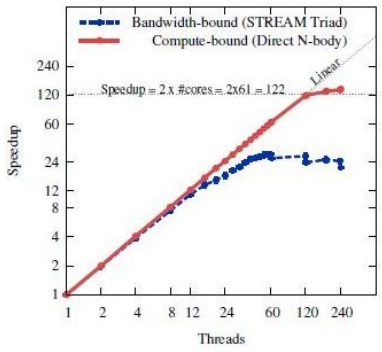
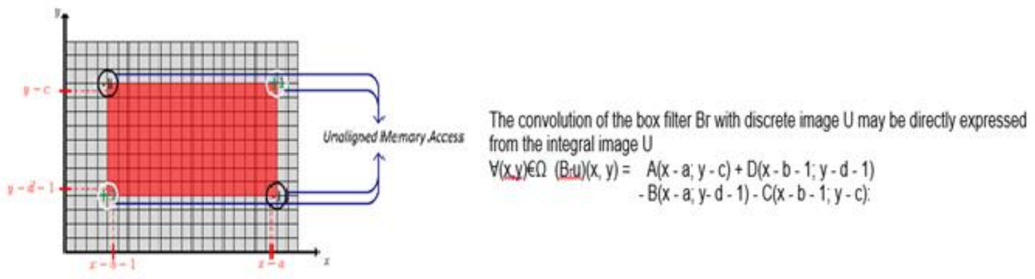


Fig 6: Stream Benchmark performance of Xeon Phi

The SURF algorithm IPOL[18] and SURF[9] are two open source SURF implementations on which we base our analysis and implementations. SURF C++ implementation is used in our analysis for Intel Xeon Phi. We initiate with porting the legacy SURF C++ code[14] on Intel Xeon along with OpenCV and Intel Compiler dependencies. We have three programming models to evaluate for Intel Xeon Phi: Offload OpenCL, Offload OpenMP and Native OpenMP. It requires that OpenCV library should be built for Intel Xeon E5 host and Intel Xeon Phi (native) which due to lack of root access and OpenCV library dependencies was an iterative and time consuming process. Initially we were able to port OpenCV for the offload model followed by native model in another few weeks' time. SURF algorithm is not only have an algorithmic complexity of O(N), but also have lot of unaligned and un-coalesced memory accesses (due to BOX integral (Fig 7) and scale space pyramid). This nature of SURF kernel makes it tricky to extract performance from the Intel Xeon Phi platform.



The convolution of the box filter Br with discrete image U may be directly expressed from the integral image U

$\forall (x,y) \in \Omega \quad (B_r u)(x, y) = A(x - a; y - c) + D(x - b - 1; y - d - 1) - B(x - a; y - d - 1) - C(x - b - 1; y - c);$

Fig 7: BOX Integral memory access pattern

The SURF algorithm could be divided into five kernels: Integral Image computation, Constructing Scale Space Pyramid, Feature Detection, Orientation Assignment, and computing Feature descriptors. We profiled the SURF algorithm to get the time distribution of each of the kernels in the total time taken by the SURF algorithm to execute serially on Intel Xeon E5 (for Offload models) and Intel Xeon Phi (for Native model). We use the SURF OpenCL [19] and C++[18] implementation for correctness for serial baselines. So we planned to modify those kernels which occupied the biggest chunk of the total time taken by the all of them together. Thus in order to parallelize the algorithm our target became two kernels namely Orientation Assignment, and Computing

Feature descriptors as these were currently occupying around 60% of the total time (Fig 9) and this time will increase as the number of descriptors scale, using the various optimization techniques suggested in the[6] for the Intel Xeon Phi.

We planned to look at Multithreading optimization followed by Vectorization and keeping tabs with memory Access patterns. We added the scalar optimizations later which enhance the performance extracted from the other optimizations. For the offload model it also becomes necessary to keep tab on communication overhead. The optimization was approached keeping in mind the fact the memory bandwidth (due to Ring interconnect) is the bottleneck for Intel Xeon Phi Knights Corner so we don't look for 20X gains as claimed by the SURF[9] OpenCL for GPUs. We approached the problem by analyzing the constraints of each of the programming models (OpenCL Offload, OpenMP Offload and OpenMP Native shown in Table 1 and aiming at not more than single digit gains in comparison to their respective serial host codes.

Table 1: Various Programming models based on Host and Accelerator

Programming Model	Host	Accelerator
Offload OpenCL	Intel Xeon E5	Intel Xeon Phi 5110P
Offload OpenMP	Intel Xeon E5	Intel Xeon Phi 5110P
Native OpenMP	Intel Xeon Phi 5110P	Intel Xeon Phi 5110P

Since we aim to implement the feature matching application for three classes of images: Bird, Car, Watercraft using the SURF feature extraction and feature matching, we took 100 images of various resolutions from ImageNet database for our application analysis. There are 30 images from each class and 10 images not belonging to any class. The SURF kernel and feature matching are the two major components of this application. We concentrate majorly on SURF kernel optimization as feature matching becomes a bottleneck only when we scale the dataset of images over thousands of images. The feature matching is currently implemented using brute force least Euclidean distance feature matching. We also observe the effect of the SURF kernel optimization on the feature matching application and analyze the scope of further speed up for the application.

Experiments

Kernel Constraints

a) Box Integral computation is used for calculating the Haar wavelets X and Y for Orientation Assignment, Calculating Feature Descriptors, Fast Hessian Kernel (Hessian determinant kernel) which on profiling tend to dominate the total execution time of SURF Kernel. Each Box Integral computation access four pixel values and for each HaarX and HaarY we compute two Box Integrals so total eight pixel access. These single pixel accesses would be mostly unaligned and un-coalesced.

b) The dominating part of the application: Orientation Assignment and Calculating Feature Descriptors are called for each interest points. The total number of internal loops in Orientation Assignment and Calculating Feature Descriptor kernels are 169 and 16x81 = 1296 respectively. They both have complexity O(N) where N is given by number of loops multiplied by number of descriptors. Number of descriptors varies for each image. Both the kernel have reduction sum operation for calculating orientation and descriptors with the internal loops iterations not a multiple of 16. This is a constraint as Xeon Phi have 512 -bit SIMD lanes so 16 4-byte single precision.

c) The Fast Hessian kernel also dominates time distribution of the SURF algorithm mainly due to two reasons: unaligned memory accesses in 6 Box Integral calls per Hessian determinant along with un-coalesced due to varied filter sizes of scale space pyramid. Note that fast Hessian kernel has one additional limitation for offload model that it generates number of interest points and in this case array buffer used for offloading data may overflow as relocatable C++ structures like vectors are not allowed in the offload model.

d) Our application is bandwidth-bound with a complexity of order of O(N) from the figure 6 referenced from [6], we observe that our application could be similar to the STREAM triad. This figure gives us an indication there should be sweet spot for the number of cores for our application. For the STREAM triad that sweet spot was found around 24. On experiments with our SURF kernel on Intel Xeon Phi we found the sweet spot around 36 cores. The number of cores is modulated by setting **OMP_NUM_THREADS** variable equivalent to the number of threads required to be scheduled.

Multithreading and Memory Access

OpenMP(Native and Offload):Foremost since the computation time of SURF algorithm is determined by the number of interest points so the multithreading is applied for the number of interest points for calculations of interest point orientations and descriptors.[6] For OpenMP multithreading it is suggested to use to modify the thread scheduling using static, dynamic and guided.

Static scheduling is dividing the loop iterations as equal as possible among the threads. In dynamic scheduling the internal work queue is used to give chunk size block of loop iterations to each thread. Guided scheduling is similar to dynamic but initially the chunk size is large and then decreases to handle the load imbalance. We tried these schedule mode as experiments in our multithreaded scheduling. It is noteworthy that in OpenMP Offload has a limit that the Offload does not work on C++ constructs, while there is no such limitation in OpenMP Native model. So all std::vector, classes and other std functions were required to be converted to respective C functions and constructs for allowing to be offloaded. For offload OpenMP we have to add extra communication over PCI express from host to Xeon Phi compared to native implementation. It is to be noted that in order to minimize unaligned access all the dynamic buffer are allocated using aligned malloc and static buffers are explicitly aligned to 64 byte.

OpenCL: [19] The number of workgroups should be bigger than the number of compute units or at least equal to the compute units and suggested size is greater than 1000.To achieve better performance and parallelism between workgroups, it should be kept in mind that the workgroup execution takes more than 100000 clock cycles. We schedule the work-group size at the least on the basis of the number of descriptors which is normally close to 1000 or even more. It is suggested to have memory map buffers for communication between host and devices. OpenCL has a control to select images for processing data instead of data buffers, while for Intel Xeon Phi data buffers are suggested to have lower latency than images. For our kernel we utilize data buffers and mapped memory buffers between host and devices these optimizations. We also use structures of arrays instead of array of structures like orientation and descriptors for all interest points which increases the memory coalescing.

Calculating Feature descriptors

The feature descriptors is 64-bit wide float vector which is calculated for each interest points. The window of 20s size is actually divided into 4x4 sub regions with Haar wavelet responses calculated over 9x9 iterations. We experiment with the first approach of the SURF kernel suggested by SURF OpenCL [9] which creates two kernel functions first for computing the Haar wavelet responses for 9x9 summing each sub regions and then second kernel which implements the reduction over the length and normalize the descriptors.

OpenMP (Native & Offload): Each descriptor runs independently on a single thread which should reduce the time a lot and additional gains are expected from vectorization for a single thread. The initial serial code works on a while loop computing i and j value each iteration, we convert this to for loop with i and j precomputed values to know the scope of vectorization. Moreover, we split the internal 9x9 for loop into two parts firstly we compute the four values dx, dy, mdx and mdy for an array of 81 and then apply reduction sum with vector dependence on that array . We use the Intel specific exponent function calls for Gaussian kernel value computation as it is suggested the specific mathematical functions are more optimized for Intel architecture. This leads to removal of vector dependence in the internal loop but due to gather and indirect access of Box Integral and peel loops the vectorization gains detected by the compiler in only 2.5X. We also split the calculation of the normalized length of the descriptor outside of this loop and followed by a separate loop of descriptor normalization. The limitations of this implementation is using array of structures for interest points so the descriptor array across threads is not coalesced.

OpenCL: Each workgroup size 9x9 = 81 workgroup size with 16 workgroup for each descriptors which ensured by keeping a single buffer for descriptors of all interest points. The total number of workgroups would be equal to number of descriptor points x16. The descriptor buffer is common for all the descriptor points and is the size of the number of workgroups. For the outer loop size of each 4x4 sub-regions, we precompute the i and j values a 16 float value two buffers to allow vectorization and predetermined workgroup sizes. The HaarX and HaarY are calculated using Box Integral and for each 81 values and multiplying it with Gaussian response of 2.5 times the scale deviation. After this step we have 4x81 values for all 16 values we do the reduction sum for each of the 81 values. The explicit vectorization is avoided to confuse the compiler and auto-vectorization is the suggested approach. Though with vectorized length being not a multiple of 16 it is definitely would generate peel loops. The 64 valued descriptor has the processed by reduction sum in the normalized descriptor kernel to compute the normalized length value and then the descriptor are normalized by the length value. To ensure vectorization the workgroup size in normalized descriptor computation we set a workgroup size of 64 which the size of descriptor size. Though the normalized descriptor kernel is vectorized but has less computation and reduction , while the create descriptor kernel have much higher computation, but a non-multiple 16 inner loop and gather memory access due to un-coalesced and unaligned access pattern could hamper the gains.

Orientation Assignment

OpenMP (Native & Offload): We actually break the computations of the getOrientation() into three parts. We precompute the dual loop variable of 109 sizes and use it for the computation of resx and resy computations. This constructs a 109 size buffers of resx and resy and removes loop interdependence which prevents vectorization. We use the Intel specific exponent function calls for

Gaussian kernel value computation instead of look up table for the gaussian values, as it is suggested the specific mathematical functions are more optimized for Intel architecture. The compiler report indicates 3.5X expected gains due to above vectorization and removal of interdependence. It also indicated that there are gather and indirect access for box integral called by HaarX and HaarY. The resX and resY value computation are followed by the angle computation which has arc tangent computation which is again using Intel specific mathematical functions. The getangle() function calls need to be vectorized by giving vector (uniform) directive for the function calls which will indicates the compiler to treat it as elemental function for vectorization of the loops. The loop splitting of the angle computation gives us vectorization, though the loop iterations not being a multiple of 16 will generate a peel loop. The third part of the kernel optimization is similar to OpenCL optimization in the inner 109 loop branch is removed and max operation for the calculation of the dominant orientation prevents loop fusion.

OpenCL: We distribute the getOrientation() into two kernels first have angle calculation and other orientation calculation. The first kernel loop has 169 iterations in total in which total 109 iterations will be masked and total result is stored in 109 sized resX, resY and angle buffers. This kernel is expected to materialize into vector masked kernel operations and for vectorization the internal loop size is kept to be 169. The second kernel for orientation assignment calculates the orientation over a window that slides by pi/3 each time for 42 times and inner loop of 109 values has a horizontal summation over resx and resy values. We don't split the resX and resY summation across the two branch condition in the serial code instead we merge the two condition by logical compares to avoid branching to be assumed which may prevent vectorization. The inner loop removal of branch increase the chance of vectorization and expected vectorization gains. The inner loop of 109 is followed by calculate the max orientation which is done each time for 42 vectors to calculate the dominant orientation. This max operation prevents us from merging inner and outer loop as it is loop interdependent operation and would lead to no vectorization even in the 109 iterations of the inner loop currently available.

Fast Hessian Kernel (Constructing Scale Space + Feature Detection)

Getting interest points using scale space pyramid calculation of hessian Determinant and followed by selection of interest points (feature detection) by non-maximum suppression. This kernel is observed to take a large percentage of the time taken for the SURF kernel for both Xeon E5 host and Xeon Phi (Native host). This could be attributed to the fact that un-coalesced and aligned memory accesses in its implementation which will leads to regular memory access from the main memory. This behavior of the memory access is a problem for overall algorithm as we know that for Xeon phi the memory access can be a bottleneck. We tried to parallelize the create response map in scale space construction by multithreading on the respective host for each programming model (Fig 9) expecting multithreading will alleviate some memory access losses using OpenMP directive. It is also assumed initially that fast hessian kernel would remain the same as number of interest points increase for an image. We do expect from serial profiling it should be the focus of optimization after the optimization in getOrientation () and getDescriptor() kernels.

Integral Image

The integral image is the first stage of the SURF kernel and is actually the accumulation of the pixel values over the images from top left to bottom right side of the image. The integral image is essentially a two prefix scan one across width and other across height. As suggested in [9], this can be implemented by a prefix scan over width followed by a transpose then a prefix scan along width followed by a transpose. It is to be noted that Integral image currently is the least dominant kernel in serial baseline profiling, so we have not focused on its optimization in either of the programming models.

Compiler Optimizations

There are various scalar optimizations which can be applied to get best possible gains on the system. Some of these environment variables are KMP_AFFINITY, Aliasing, Prefetching and Scheduling. KMP_AFFINITY: Threads were set in compact, scatter and balanced mode to see their effect on parallelization. Aliasing: Due to aliasing Intel compiler takes cautious view to vectorize a loop, which sometimes result in loop not being vectorized. So we used flag –fno-alias in order to remove aliasing. Loop Scheduling: It could be scheduled as Static, Dynamic, Guided and Auto. We tried all the four types to see their effect on our results. Prefetching: Memory load latencies can be reduced drastically using a technique known as prefetching. Prefetching is turned on by default for Intel Compilers using OpenMP if optimization level is kept -O2 or above. In OpenCL model we explicitly prefetched data manually.

Feature Matching application

The application has three basic steps: Creating Dictionary for three classes of descriptors and then read a test image, creating SURF feature descriptors for test image, and matching the test image with the three classes and store the matched class. The problem with SURF feature matching is that the matches would match if we observe the exact feature descriptors of the object in the dictionary. The feature matcher just matches the features of the test image and three dictionary images by least Euclidean distance. To have a match in a test image for any dictionary feature descriptors of class images one of the example is shown in fig 12(a), that exact image or a major portion should be part of the test image. This criteria makes it essential to matching which won't work for general class of flowers, birds, and cars, but will work for same image in our flowers class as our reference image. Thus we ensure some test images contains the reference dictionary object, while others contains other from that classes. Various stages of feature matching application is shown below in Fig 8.

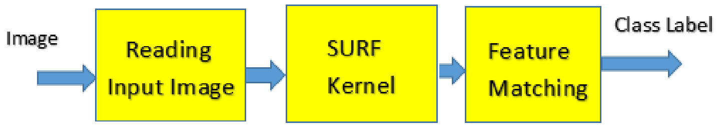
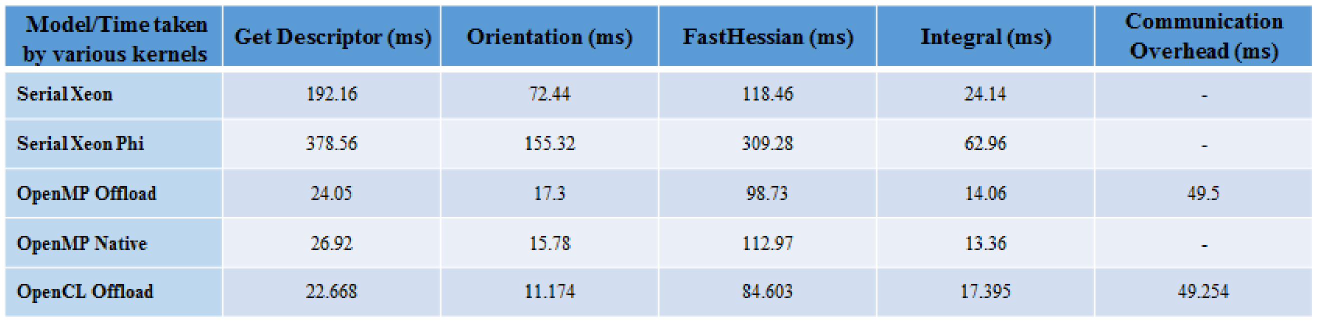


Fig 8: Various stages of Application

Results

Serial SURF profiling on Intel Xeon

After resolving all the dependencies, installing OpenCV and porting SURF C++ legacy code on Xeon E5 and Xeon Phi 5110P CPU hosts for offload and native programming models respectively. From the results shown in Table 2 and pie chart 9 we saw that major portion of computation time for a serial code on respective hosts was taken by kernels Get Descriptor, Fast Hessian and Orientation. An important observation is that the total time taken by Xeon Phi coprocessor host to run code natively was almost twice as much as time taken by Xeon E5 host. This indicates two things that a single core of Intel Xeon E5 is more powerful than a single core on Intel Xeon Phi and irregular memory access from the global memory, as in Fast Hessian kernel, are more costly on Intel Xeon Phi than Intel Xeon E5.From the serial profiling results, it is indicated Getdescriptor () should be targeted, and Orientation () also scales on the basis of interest points as Getdescriptor(), so optimize both the kernels in tandem.



Model/Time taken by various kernels	Get Descriptor (ms)	Orientation (ms)	FastHessian (ms)	Integral (ms)	Communication Overhead (ms)
Serial Xeon	192.16	72.44	118.46	24.14	-
Serial Xeon Phi	378.56	155.32	309.28	62.96	-
OpenMP Offload	24.05	17.3	98.73	14.06	49.5
OpenMP Native	26.92	15.78	112.97	13.36	-
OpenCL Offload	22.668	11.174	84.603	17.395	49.254

Table 2: Execution times of SURF kernels during serial execution on Intel Xeon E5, Intel Xeon Phi as hosts, respectively. The number are best serial case numbers observed .Execution times of SURF kernels during parallel execution in OpenMP Offload model with Xeon as host and Xeon Phi as accelerator, OpenMP Native model with Xeon Phi as host as well as accelerator and OpenCL Offload model with Xeon as host and Xeon Phi as accelerator

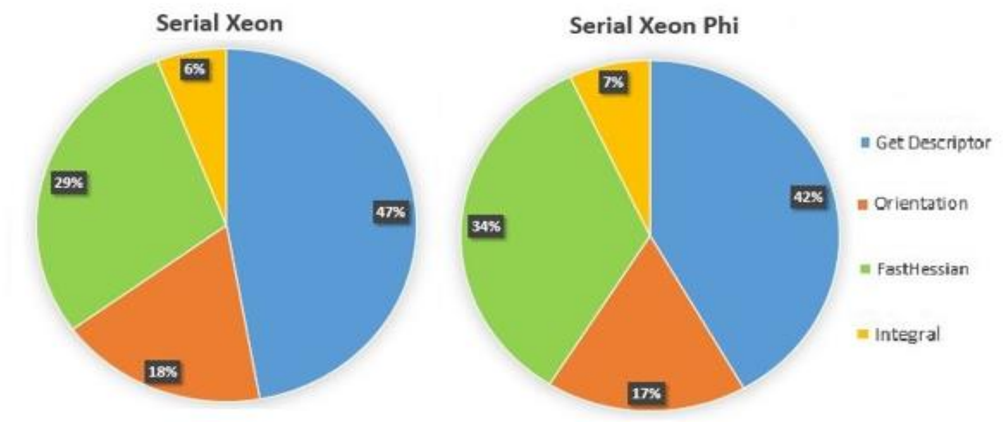


Fig 9: Pie Chart depiction of time distribution of SURF kernels during serial execution on Xeon and Xeon Phi as hosts

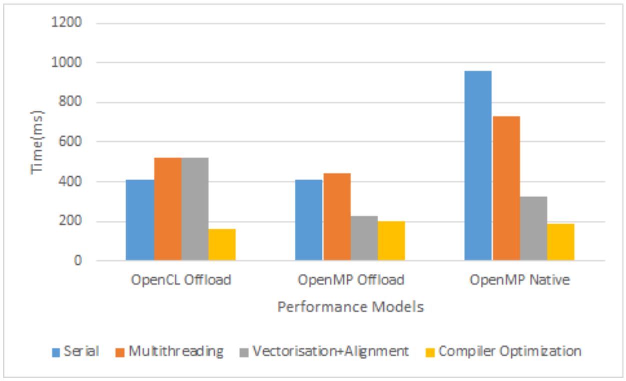


Fig 10: Effect on execution timing of code after various optimizations are applied. Note that scalar optimizations is added optimizations over multithreading, vectorization & alignment and helps to extract the actual gains targeted by multithreading and vectorizations.

In order to take advantage of Xeon Phi coprocessors we offloaded code and respective data on to the Xeon Phi cores. OpenMP(Offload & Native): (Fig 9) Multithreading in getDescriptors() and getOrientations() is offloaded to Intel Xeon Phi on the 60 cores we observed an unexpected dip in performance for the algorithm for Offload in OpenMP. Multithreading in getDescriptors() and getOrientations() happens on Intel Xeon Phi on the 60 cores we observed some gains with respective to native serial implementation. We also multithreaded the buildResponsemap() of the Fast Hessian kernel though only on Intel Xeon E5 host and Intel Xeon Phi host for respective models. We observe that execution time of serial code running on Intel Xeon Phi is much higher than serial code execution time on Intel Xeon E5 host. This dip in performance could be attributed to the fact that due to the simultaneous accesses in multithreading, the memory access bandwidth requirement from the main memory increases drastically for Intel Xeon Phi. The memory access are unaligned access of single FP data by each thread which leads to repeated main memory accesses for the same memory location. The feature descriptor points/Interest points are normally concentrated around regions of an image not evenly distributed which leads to repeated memory access to same memory location in the integral image. This exerts a traffic bottleneck on the Ring Interconnect of the Intel Xeon Phi, so it was required to minimize the memory accesses. For this memory access needs to be aligned and un-coalesced. By aligning the buffer memory access and vectorization some single memory accesses are converted to SIMD memory access at aligned memory boundaries. Though still the gains are not as per our expectation, so we look into some compiler optimizations. The thread affinity was set to scatter as SURF is a bandwidth bound application. Also various combinations for the number of threads were targeted. We found sweet spot around 36 cores and 60 cores in offload and native models respectively. We additionally introduce -fno-alias which instructs the Intel compiler to allow vectorization when accessing the same memory location by different pointer spread across the threads or within the thread. The compiler in OpenMP adds automatic prefetching at optimization level -O2 or higher.

Though we had an intuition to have considerable gains in OpenMP offload with respect to Xeon E5 serial code but results were less than expected, this doubt was resolved by the profiling analysis of the optimized SURF which showed only 2.1x gains. The Pie charts in Fig 11(b) of OpenMP indicate that communication overload has considerably increased for the offload. This could be attributed to the transfer of integral image and memory chunk of the interest points. The fast hessian kernel now becomes the dominant kernel in SURF algorithm as expected in spite of multithreading in the buildResponsemap(). This could be attributed to the fact that the memory access are still unaligned and un-coalesced in Fast Hessian and currently using C++ constructs and buffers. The integral image kernel also shows a higher percentage as the numbers in other kernel gets reduced.

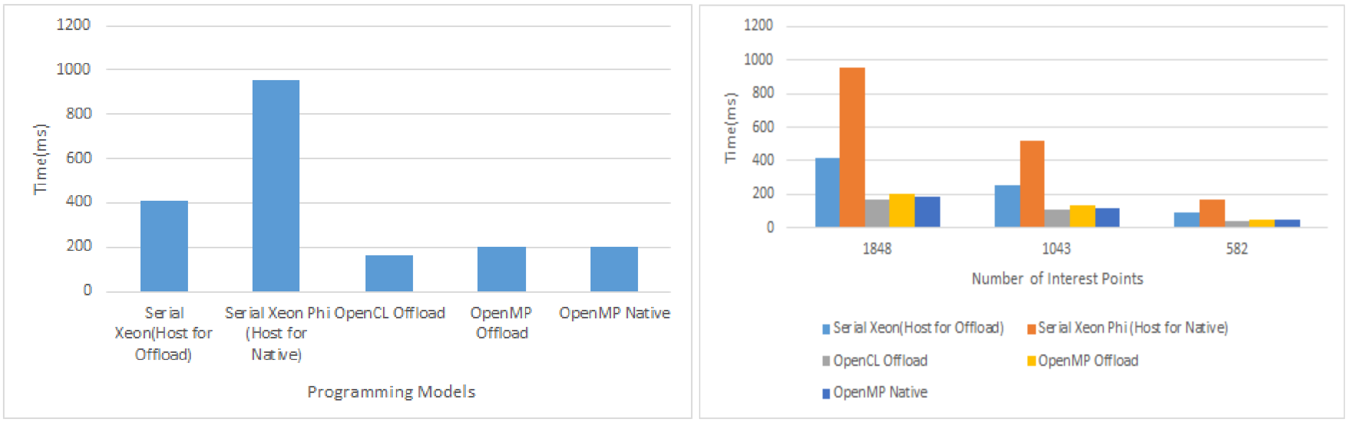


Fig 10: (a) Execution time for various programming models, Number of interest points are 1848
(b) Execution time for various models based on number of Interest points

Though we expected substantial speedup but figure of 2.2x compared to Serial Xeon E5 and 4x in comparison to Xeon Phi Serial code execution are less than expected for Native OpenMP model. Our doubt was resolved again by the results of profiling analysis done for optimized SURF Native kernel. The Pie charts in Fig 11(c) of OpenMP Native indicate the Fast Hessian kernel is sole dominant occupant in the time taken according to the pie chart. We know that Fast Hessian kernel have a lot of aligned and un-coalesced memory accesses as was with OpenMP Offload model also, but just a higher percentage was surprising. The fast hessian kernel is also mostly single threaded except buildResponselayermap(). This makes Fast Hessian our next kernel to be targeted in future to extract desired level of performance from this SURF algorithm in native model.

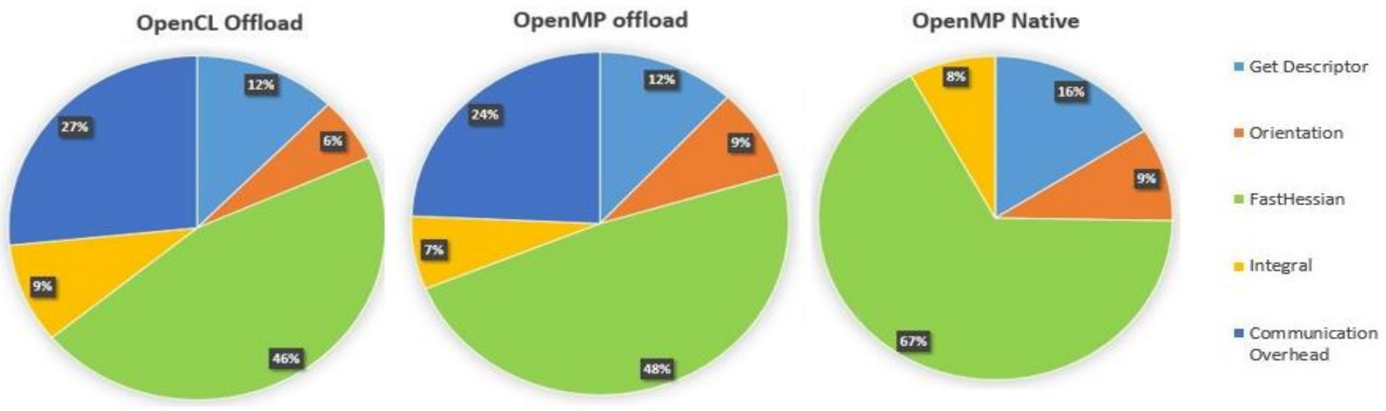


Fig 11: Pie Chart depiction of time distribution of SURF kernels parallel execution in (a)OpenCL Offload, (b)OpenMP Offload model with Xeon as host and Xeon Phi as accelerator, (c)OpenMP Native model with Xeon Phi as host as well as accelerator

OpenCL (Offload Model): Multithreading and vectorization alignment numbers can't be separated for OpenCL implementations hence both optimizations were targeted simultaneously on OpenCL model. So actually red and grey tabs shown in Fig 9 under OpenCL Offload, combined delivers the performance. It is to be noted that multithreading and vectorization gains were completely masked out until we specified level 2 prefetch specifically for the OpenCL kernels while compilation. The OpenCL experienced gains of approx. 2.6X over serial Xeon E5 post optimization. The multithreading, vectorization and memory alignment were done only for getDescriptors() and getOrientations() kernels,. The remaining two kernels of fast hessian kernel and integral image behave in the similar fashion in OpenMP offload.The Pie charts in Fig 11(a) of OpenCL Offload shows a similar trend as OpenMP Offload where communication and Fast Hessian kernel are primary occupants. . In the OpenCL model the communication overhead is incurred at the last point when fetching the final results from the memory mainly and other overheads include configuring the kernels from the host. For communication, mapped buffers with non-blocking access were used, still such increase is slightly surprising. The Fast Hessian kernel will be the next target for OpenCL model as well same reason as OpenMP implementation. We were able to Structure of arrays in OpenCL for all data buffers used in the descriptor calculations for OpenCL, but was not able to do the same of OpenMP model. So for future scope, we could attempt to implement structure of arrays for memory coalesced. This may be one reason why the gains in OpenCl are more than both OpenMP models

The programming model of offload (OpenCL and OpenMP) incur a high communication overhead. The bottleneck and single threaded kernels are more costly on Intel Xeon Phi (Native) than Intel Xeon E5 host. The aligned and coalesced memory access is a bigger bottleneck in Intel Xeon Phi than Intel Xeon E5. As the number of cores increases simultaneously memory accesses also increases in Xeon Phi, in turn the memory bandwidth issues compound further. The algorithm and code for OpenMP both native model is same besides the communication part, and the impact of which is seen in the first call for getorientation(). To reduce this offload communication, it needs to be done while before they are used in the offload model. The OpenCL model implements a similar algorithm but with completely different coding styles. OpenCL is not performance portable and can't work on Native. OpenMP codes are performance portable across Intel Architectures even in native and offload model Intel Xeon Phi Model.

The Fig 10(b) based on number of Interest points just shows that higher the number of interest points the higher the serial code execution time numbers, but after parallelization this dependency goes away as both the kernels depending upon the interest points are relatively less dominant now.

Feature Matching Application

We expect that matching would work for only exact dictionary objects, which came out to be correct. Profiling of execution timing of serial as well as parallel code is given below under table 3. We totally use 100 images from the database [20] , in which have 10 images out of 30 for each class which has the exact dictionary object or major portion of the dictionary, and we observe the correct class matching for those objects only. This suggests that we need to have a more sophisticated feature matcher or we could increase class matcher by using a classifier like SVM. The SURF algorithm individual gains directly don't translate into the application gains. The matching kernel is serialized and takes more time as number of descriptors increase. Parallelizing the feature matching kernel and pipelining the three stages of our application would extract further performance gains, which will be our future goal to modify along with Fast hessian kernel of SURF algorithm. Rest of the reference and resultant images can be seen under Appendix given under Fig 13, 14 and 15.

	Feature Matching Application (ms)
Serial Xeon	15.53
Serial Xeon Phi	84.27
OpenMP Offload	12.53
OpenMP Native	60.23
OpenCL Offload	12.07

Table 3: Total Execution time of Feature Matching Application for various models serial as well as parallel. In Offload model Xeon E5 acts as host and Xeon Phi 5110p as accelerator, In Native model Xeon Phi 5110p acts as host as well as accelerator

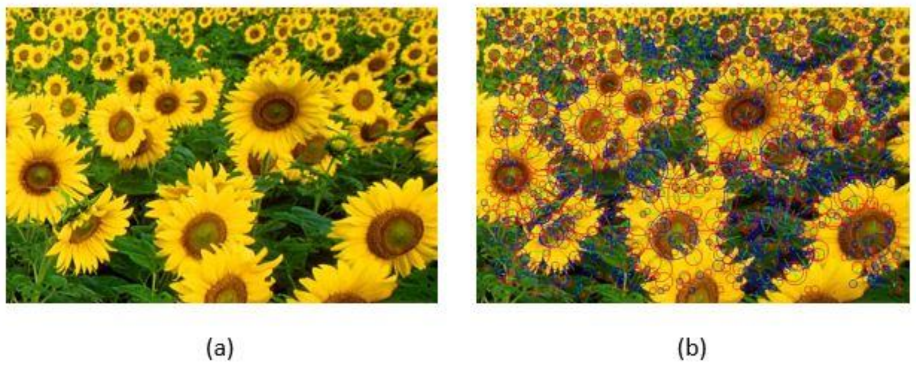


Fig 12: a) Reference Image b) Resultant Image after running the feature matching application

Conclusion

Intel Xeon Phi is an interesting and exciting platform for high Performance Computing domain. It suffers from memory bandwidth issues due to its Ring interconnect which increases further as multithreading increases. It is to be noted that it does not matter how much multithreaded or vectorization optimizations are done, until memory access are made aligned and coalesced one won't be to extract much performance gains from Intel Xeon Phi. It is also to be noted that for an application like feature matching for large datasets it would be better to parallelize the matching kernels as the matching kernels is compute bound kernel. Moreover, it is also to be noted that the internal loops which need to be vectorized for an application should have parallelization to exploit 512-bit SIMD lanes. To exploit these gains, the internal loops should not have any sort of loop interdependence. SURF kernel is not only bandwidth bound with gather accesses, but also small internal loops or loops with interdependence. This suggests that SURF kernel is not a desired kernel to be optimized for Intel Xeon Phi. But still if one wants to optimize further the SURF kernel one further requires the focus to be on Fast Hessian kernel followed by Integral Image.

The Offload programming model of OpenCL and OpenMP have a high communication overhead compared to Native. It also should be taken into account that penalty of a serialized code on Intel Xeon Phi 5110p is much more than on Intel Xeon E5. This is due to the fact that a single core of Intel Xeon Phi is not as powerful as a single core of Intel Xeon E5. The Native model implementation suffers from this penalty. OpenCL code development is tougher, and more intensive than OpenMP, but more inclined to extract parallelization. OpenCL code though is portable on other non-Intel platforms, but performance portable cannot be guaranteed. OpenMP compiler on Intel Xeon Phi is much more intelligent and suggests many bottlenecks and optimizations in optreport. OpenMP (Native and Offload) are more preferred programming models for Intel Xeon Phi development. The choice between Native Model is preferred for application with high coalesced data exchange among kernels. The Offload should be preferred for applications having portions of serialized code and unaligned memory access. It could be concluded that Intel Xeon Phi is an attractive HPC platform, but the effort for parallelization due to high width SIMD lanes and limited memory bandwidth is higher.

Acknowledgment

The satisfaction that accompanies the successful completion of any task would be incomplete without mentioning about the people who made it possible. We are highly indebted to **Dr. Alan D. George** whose constant support and guidance helped us to complete the project successfully. The classes and the material provided on the parallel programming languages especially for material for Xeon and Xeon Phi were also very useful that helped us a great deal in completing our project. Our TA's **Dhananjay Kumar** and **Vandana Ravichandran** were always ready to give us a helping hand. Another great support were the Parallel Programming Lectures and training on Intel Xeon Phi provided by Coalfax Research International.

References

[1] *Intel® Many Integrated Core Architecture – Advanced* Here

[2] Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool. *"Surf: Speeded up robust features."* Computer vision–ECCV 2006. Springer Berlin Heidelberg, 2006. 404-417 Here

[3] Jeffers, James, and James Reinders. Intel Xeon Phi coprocessor high-performance programming. Newnes, 2013.

[4] Is Next-Gen Xeon Phi a Supercomputer-GPU Killer? Here

[5] Leung, Kai-Cheung, et al. *"Investigating large-scale feature matching using the Intel® Xeon Phi™ coprocessor."* Image and Vision Computing New Zealand (IVCNZ), 2013 28th International Conference of. IEEE, 2013.

[6] Coalfax Research International Training Here

[7] Calvin, Christophe, Fan Ye, and Serge Petiton. *"The exploration of pervasive and fine-grained parallel model applied on Intel Xeon Phi coprocessor"* ; P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on IEEE, 2013. Here

[8] *OpenMP* Thread Affinity Control*.**Here**

[9] Yan, Wanglong, et al. *"Computing SURF on OpenCL and general purpose GPU"*; International Journal of Advanced Robotic Systems 10 (2013).

[10] Optimization and Performance Tuning for Intel® Xeon Phi™ Coprocessors - Part 1: Optimization Essentials Here

[11] Dolbeau, Romain, François Bodin, and Guillaume Colin de Verdiere, *"One OpenCL to rule them all?"* Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on IEEE, 2013.

[13] *Optimizing OpenCL Applications on Intel® Xeon Phi™ Coprocessor*.Here

[14] SURF C++ Code Here

[15] *Optimizing Memory Bandwidth on Stream Triad***Here**

[16] Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture.Here

[17] Intel's 50-core champion: In-depth on Xeon Phi.Here

[18] Oyallon, Edouard, and Julien Rabin. "An Analysis of the SURF Method."Image Processing On Line 5 (2015): 176-218. Here

[19] Threading: Achieving Parallelism Between Work-Groups Here

[20] http://vision.cs.unc.edu/ilsvrc2015/ILSVRC2015_VID_snippets.tar.gz

Appendix

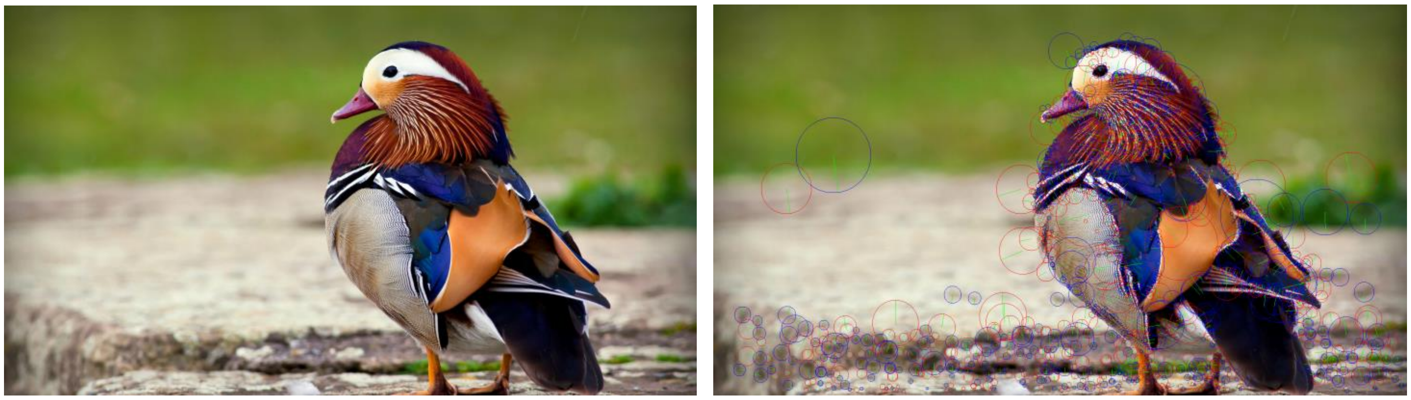


Fig 13: a) Reference Image b) Resultant Image after running feature matching application

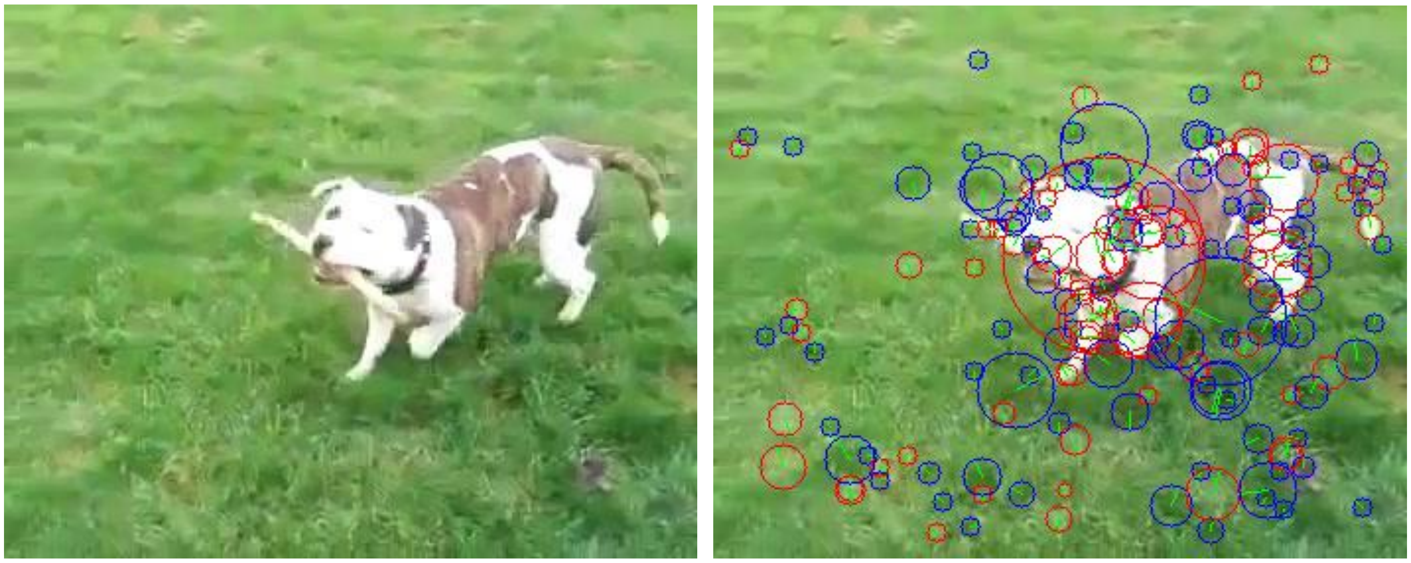


Fig 14: a) Reference Image b) Resultant Image after running feature matching application

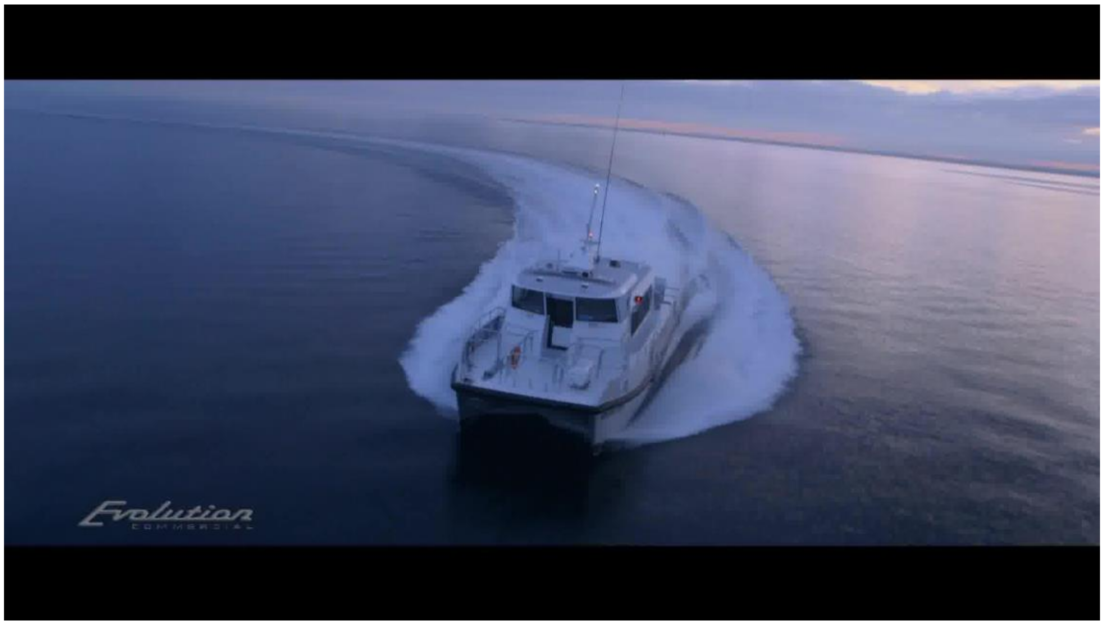


Fig 15: a) Reference image of watercraft

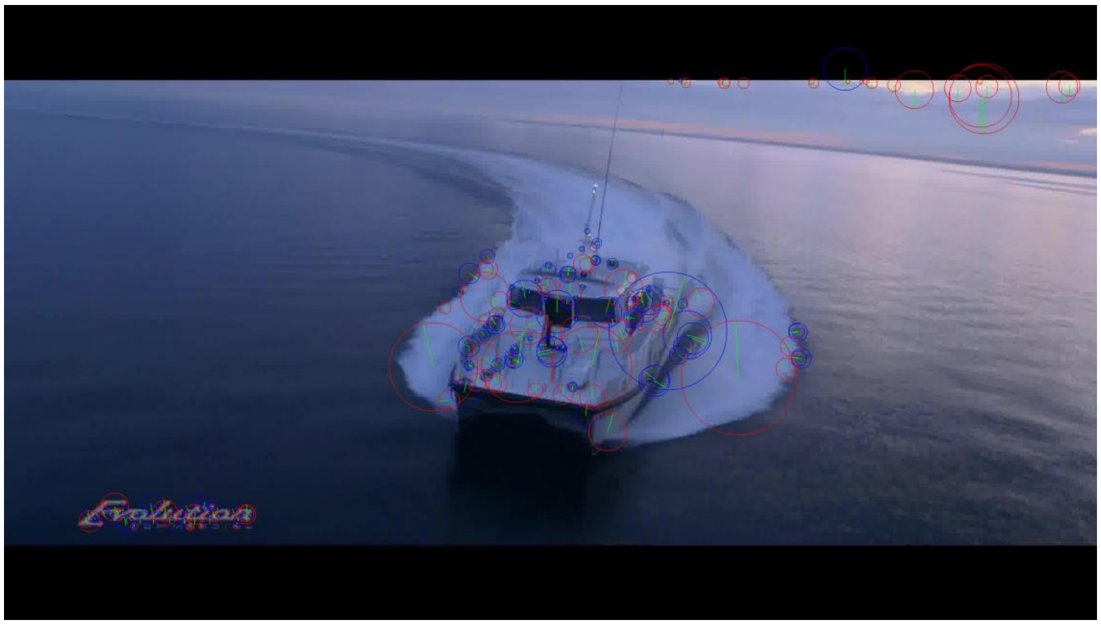


Fig 15: b) Resultant image of watercraft after running feature matching application