Performance Evaluation of Medical Imaging Algorithms on Intel® MIC Platform

Jyotsna Khemka*, Mrugesh Gajjar*, Sharan Vaswani*, Naga Vydyanathan*,
Rama Malladi† and Vinutha S V†
*Siemens Corporate Research and Technology Center, Bangalore, India 560100
{jyotsna.khemka,mrugesh.gajjar,sharan.vaswani,nagavijayalakshmi.vydyanathan}@siemens.com
†Intel Technology India Pvt. Ltd., Bangalore, India 560017
{rama.kishan.v.malladi,vinutha.s.v}@intel.com

***Abstract*—Heterogeneous computer architectures, where CPUs co-exist with accelerators such as vector coprocessors, GPUs and FPGAs, are rapidly evolving to be powerful platforms for tomorrow's exa-scale computing. The Intel® Many Integrated Core (MIC) architecture is Intel's first step towards heterogeneous computing. This paper investigates the performance of the MIC platform in the context of medical imaging and signal processing. Specifically, we analyze the achieved performance of two popular algorithms: Complex Finite Impulse Response (FIR) filtering which is used in ultrasound signal processing and Simultaneous Algebraic Reconstruction Technique (SART) which is used in 3D Computed tomography (CT) volume reconstruction. These algorithms are evaluated on Intel® Xeon Phi™ using Intel's heterogeneous offload model. Our analysis indicates that execution times of both of these algorithms are dominated by the memory access times and hence effective cache utilization as well as vectorization play a significant role in determining the achieved performance. Overall, we perceive that Intel® MIC is an easy-to-program accelerator of the future that shows good potential in terms of performance.**

***Index Terms*—Intel MIC, algebraic reconstruction algorithms, FIR, many-core, accelerators**

I. Introduction

Growing complexities, scale and performance needs of applications across all domains, be it scientific computing or business, have placed heavy demands for powerful computational resources. Though technology development has resulted in the ability to add more and more transistors on a single chip, the CPU clock speeds have hit a plateau due to several factors: heat and power barriers, memory wall, propagation delays and other physical limits. This, coupled with the limits on ILP (instruction level parallelism) as well as hardware limitations (register renaming, branch mis-predictions, instruction window sizes etc.), have led to the rise of multi-core systems and thread level parallelism (TLP) [1].

Traditionally, CPU design in multi-core systems focuses on improving single thread performance. To scale up to the computing needs of future, a new paradigm of accelerator designs focusing on throughput and increased data and thread level parallelism (TLP) has emerged. The accelerator cores are larger in number and operate at lower frequencies resulting in better power-efficiency as compared to multi-cores. Thus, heterogeneous architectures that combine general purpose multi-core CPUs with accelerators have an attractive performance-to-power ratios and are crucial players in the road to exascale and green computing [2], [3].

Intel's first commercial product for heterogeneous computing is its x86 based co-processor architecture called Many Integrated Core (MIC). The MIC architecture combines a large number of smaller x86 cores with lower single threaded performance and operating at low power, to deliver a high aggregate performance. The first instantiation of the MIC architecture, Intel® Xeon Phi™ code-named Knights Corner (KNC) was released in late 2012.

This paper evaluates the performance of the Intel MIC co-processor, specifically, the Knights Corner (KNC) platform in the context of medical imaging and signal processing. We analyze the achieved performance of two popular algorithms: Complex Finite Impulse Response (FIR) filtering which is used in ultrasound signal processing and Simultaneous Algebraic Reconstruction Technique (SART) which is used in 3D Computed tomography (CT) volume reconstruction. Our analysis indicates that execution times of both of these algorithms on MIC are dominated by the memory access times and hence effective cache utilization as well as vectorization play a significant role in determining the achieved performance. Overall, we perceive that Intel® MIC is an easy-to-program accelerator of the future that shows good potential in terms of performance.

The rest of this paper is organized as follows. Section II gives an overview of the MIC architecture and its programming model. Section III, provides details of the FIR and SART algorithms. Section IV highlights the optimization strategies used to maximize performance. Section V presents the performance results and analysis and Section VI concludes the paper. Please note that we use the terms Intel® Xeon Phi™, KNC and MIC interchangeably.

II. The Intel Many Integrated Core (MIC) Architecture and Programming Model

A. MIC Architecture

Figure 1 shows a schematic illustration of the Intel MIC architecture. The MIC co-processor cores are based on the x86 architecture. They are in-order cores operating at lower frequencies, but are more in number than traditional multi-cores and allow for high compute throughput.

978-1-4799-0730-4/13/$31.00 ©2013 IEEE

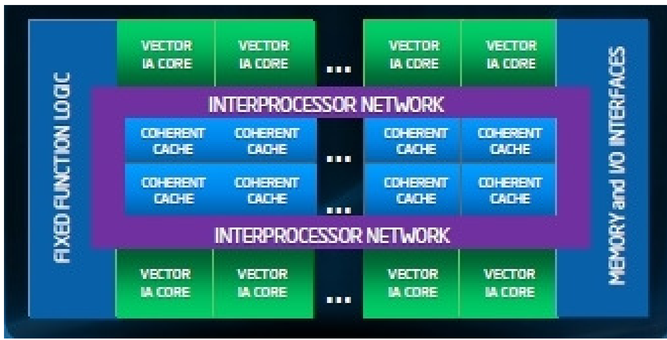
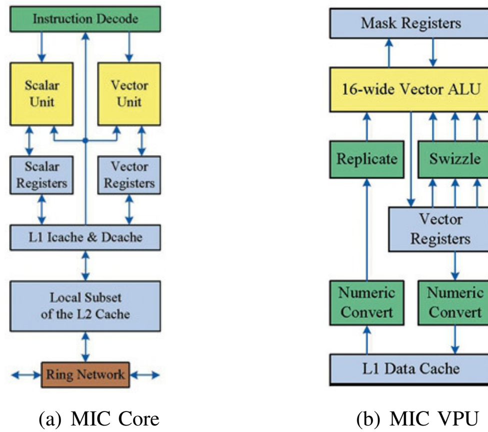


Fig. 1. The Intel MIC Architecture.



(a) MIC Core

(b) MIC VPU

Fig. 2. Block diagrams of Intel MIC core and Vector Processing Unit (VPU) [4]

The expensive x86 processor features such as out-of-order execution are stripped out in MIC (Figure 2(a)) but much of the silicon is dedicated to floating point operations. Each core runs an extended version of the x86 instruction set that includes 512 bit wide vector processing operations and is a fully functional in-order core, that supports fetch and decode instructions from four hardware thread execution contexts.

The Intel® Xeon Phi™ codenamed Knights Corner (KNC), is the first product implementation of the MIC architecture manufactured using 22nm process and 3D trigate transistors. It is implemented as a co-processor on a PCIe card with upto 61 x86 cores and peak DP performance $>$ 1 TFLOP/s. Each core has 32KB L1 instruction cache, 32KB L1 data cache and 512KB L2 cache. Cache coherency is maintained across the L2 caches by a bi-directional high bandwidth ring network, which is used to communicate between the cores, GDDR memory and PCIe. The MIC device has a fast GDDR5 memory.

An important highlight of the MIC architecture is the 512 bit wide vector processing unit (VPU)(Figure 2(b)) and the corresponding VPU instructions. In addition to standard arithmetic and logic instructions on integer and floating point data types, MIC provides fused multiply-add, gather, scatter and mask instructions. It supports up to 8 double precision (DP) or 16 single precision (SP) floating point operations within a single vector instruction.

B. MIC Programming Model

Program execution in systems with accelerators typically begins on the host CPU and some user-defined sections of the code and the corresponding data are offloaded by the CPU to the accelerators. Programming for accelerators is cumbersome and effort-intensive as compared to traditional CPU programming due to two main reasons: a) the instruction sets on the host CPU and the accelerators may not be identical, and b) the host and the accelerators do not share system memory. Mitigation of the lack of easy programmability of these systems is an active area of research.

The Intel MIC co-processor is based on x86 architecture, and hence existing code using standard parallel programming models like OpenMP [5] and MPI [6] can be easily ported to MIC. MIC supports three programming models: Native execution, heterogeneous offload programming and MPI. This paper only discusses about MIC programming using the heterogeneous offload model.

In the MIC offload programming model, the statement or the block of code following the "offload" pragma or directive is targeted for execution on MIC. An example is shown in Listing 1 in which the code section that performs the vector addition is offloaded to MIC by using "#pragma offload target(mic)". Also, please note that the code is parallelized by using OpenMP "parallel for" constructs to take advantage of the multiple co-processor cores.

```
#pragma offload target(mic)
#pragma omp parallel for default (shared)
/* vector addition operation */
for(int i=0;i<N;i++) {
  c[i] = a[i] + b[i];
} //end for i
```

Listing 1. A simple MIC program illustration

The functions and global variables to be compiled for the MIC platform have to be prefixed with a keyword "__declspec(target (mic))" or "__attribute__ ((target (mic)))". All functions in the program are always compiled for the CPU and are available to be called on the CPU. However, only functions marked with the above keyword are available to be called by MIC-offloaded code. Global variables are treated in a similar fashion. All global variables are always present in the CPU code. But only global variables with the target attribute are available on the co-processor. Compiling only functions and data explicitly marked with the target attribute into the Intel MIC architecture binary ensures that the code on the co-processor is as small as possible. For example, in Listing 2, the global arrays "input1", "input2", "output" and function "func1()" are compiled for MIC by prefixing them with the keyword "__declspec(target (mic))". However, as mentioned above, these arrays are also compiled for the CPU and are available to be called on the CPU. The function "func2()" is only compiled for CPU. When entering and leaving the offloaded "for" loop, the arrays will be copied in and out of MIC automatically. To optimize the data transfers between

MIC and CPU, we can also explicitly specify each data item with "in", "out" or "inout" qualifiers, E.g., "pragma offload target (mic) in(input1:length(100)) in (input2:length(100) out(output:length(100))".

```
__declspec ( target (mic) ) int numFloats = 100;
__declspec ( target (mic) ) float
    input1[100],input2[100];
__declspec ( target (mic) ) float output[100];
__declspec ( target (mic) ) void func1() {
  ...
}
void func2() {
  ...
}

int main() {
  read_input1();
  read_input2();
  #pragma offload target (mic)
  for(int j=0;j<numFloats;j++) {
    output[j] = input1[j] + input2[j];
  }
  func1();
  func2();
}
```

Listing 2. The MIC offload compilation of a functions and global variables

There are also clauses like "alloc_if" and "free_if" which allow persistence of memory allocation across multiple offload calls. Interested readers are encouraged to refer to Intel® Xeon Phi™ Developer's Guides [7], [4] for further details.

In addition to OpenMP, standard native parallel programming models like TBB, Cilk Plus, MKL, Pthreads, MPI are also supported on MIC.

III. Algorithms

This section gives a detailed overview of the FIR and SART algorithms. FIR filtering is widely used in ultrasound image processing to reduce gaussian and speckle noise [8], [9]. SART is an iterative algebraic reconstruction technique used in computed tomography reconstruction for obtaining good quality reconstructions with lower radiation dose [10], [11], [12].

A. Complex Finite Impulse Response Filter (FIR)

Finite Impulse Response or FIR filter [13] is one of the primary types of filters used in digital signal processing. An Nth-order filter output $y[j]$ for the current input value $x[j]$ can be expressed as the weighted sum of the current input value $x[j]$ and N previous input values as shown below,

$$y[j] = \sum_{i=0}^{N} c_i x[j-i] \tag{1}$$

where:

- $x[]$ is the input signal,
- $y[]$ is the output signal,
- c_i are the filter coefficients, also known as tap weights, that make up the impulse response,
- and N is the filter order; an Nth-order filter has $(N+1)$ summation terms

We have used a "symmetric complex FIR filter" for our evaluations. In the case of complex FIR filter, the values of input, output and filter coefficients contain both real and imaginary components.

Algorithm 1 shows the psuedo-code for applying an Nth-order symmetric complex FIR filter on a 2D image of size $ySize \times xSize$. Note that the real and imaginary values of the pixels of the input image are stored consecutively in the 1D array 'Src' and the filter coefficients are stored consecutively in the 1D array 'Coeff'. Similarly, the pixels of output image are stored in 'Dest'.

Algorithm 1 FIR

```
{Arguments: Src, Dest, Coeff, N, xSize, ySize}
{Arrays are indexed from 0}
HN=N/2;
for k=0 to ySize-1 do
  kInd=k*xSize*2
  for j=0 to xSize-1 do
    sumI=sumQ=0
    for i=0 to N do
      if (j+i-HN) ≥ 0 and (j+i-HN) < xSize then
        I=Src[kInd+ (j+i-HN)*2]
        Q=Src[kInd+ (j+i-HN)*2 + 1]
        C=Coeff[(N-i)*2]
        D=Coeff[(N-i)*2 +1]
        sumI+= I*C-Q*D
        sumQ+= Q*C+I*D
      end if
    end for
    Dest[kInd+ j*2]=sumI
    Dest[kInd+ j*2 +1]=sumQ
  end for
end for
```

B. Simultaneous Algebraic Reconstruction Technique (SART)

This section discusses the Simultaneous Algebraic Reconstruction Technique (SART) [10], [11] which is an iterative tomographic reconstruction algorithm used commonly in cone beam CT reconstruction. SART reconstructs a 2D or a 3D object from its projection data by formulating the reconstruction problem as a set of linear equations:

$$WV = P \tag{2}$$

where, V is the unknown $(N \times 1)$ vector of voxel values in the $n \times n \times n$ reconstruction grid ($N = n^3$), P is the $(R \times 1)$ vector of pixel values in the acquired projection data and W is the $(R \times N)$ weighting matrix. An element w_{ij} of the weighting matrix denotes the influence of voxel j on the ray passing through pixel i of the projection image and plays a vital role in determining the reconstructed image quality.

To solve the above equation iteratively, SART starts with an initial guess for the voxel values in the 3D reconstruction grid

and iteratively refines these values by processing the projection images. An iteration k of SART comprises of the following three steps that are applied to all the projection images:

- Forward Projection: For each pixel, p_i on a projection image, compute the line integral of the ray through p_i, i.e.,

$$\sum_{j=1}^{N} w_{ij} v_j^{(k)} \tag{3}$$

  This is illustrated in Figure 3. w_{ij} which is the weight or influence of voxel j on the ray passing through pixel i, is computed as the portion (length) of the ray that passes through voxel j using the ray traversal algorithm proposed by Jacobs et. al. [14].
- Correction Image Computation: Compute the difference between the observed pixel value p_i and the computed line integral, normalized by the weights, i.e.,

$$\frac{p_i - \sum_{j=1}^{N} w_{ij} v_j^{(k)}}{\sum_{j=1}^{N} w_{ij}} \tag{4}$$

- Backward Projection: Distributes the correction image to the reconstruction grid. We use Feldkamp's filtered back-projection algorithm [15] to back-project the correction image to the reconstruction volume.

Figure 4 depicts the cone-beam CT procedure. Cone-beam x-rays are beamed at different angles and the attenuated beam is caught on a detector plate and forms a projection image. Using several projection images, the 3D volume of the object through which the x-ray beam is transmitted, is reconstructed.

Algorithm 2 shows the pseudo code of our SART implementation which reconstructs a 3D image of size $RECON_DEPTH \times RECON_HEIGHT \times RECON_WIDTH$ from $NUM_PROJECTIONS$ 2D projection images of size $PROJECTION_HEIGHT \times PROJECTION_WIDTH$. The output 3D volume is initialized with voxel values of 1. The algorithm iteratively reads the N input projection images to "InProjData", computes the corresponding correction image data, that is, "CorrProjData" by applying the forward projection, and then distributes the correction image data to the reconstruction volume, that is to, "OutVolData" by applying the backward projection. Algorithm 3 and Algorithm 4 shows the pseudo code for our SART forward projection and backward projection implementations respectively.

IV. Implementation and Optimization on MIC

In this section we describe in detail our implementation and optimizations of FIR and SART on MIC.

A. FIR

The FIR code is parallelized to take advantage of the MIC cores using the OpenMP programming model. The parallelization is done at the level of the outermost 'for' loop so that different threads can work in parallel on different image rows.

We have implemented the following optimizations to the baseline code of FIR to minimize the compute time on MIC.

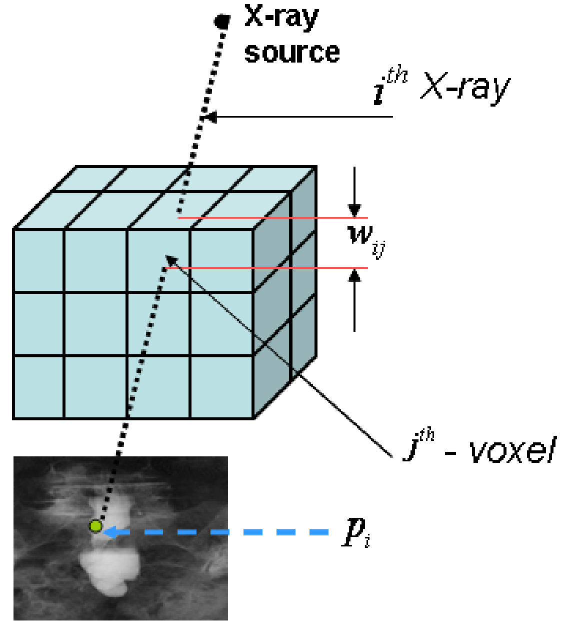


Fig. 3. Ray tracing in forward projection

Algorithm 2 SART

```
{Initialize the output 3D voxel values to 1}
InitOutputVoxels(OutVolData)
for i=1 to NUM_ITERATIONS do
  for j=1 to NUM_PROJECTIONS do
    {Get j'th projection data}
    ReadProjectionData(InputProj,j)
    {Apply forward projection for j'th projection input
    image and compute correction data}
    ForwardProjection(InputProj,OutVolData,CorrProjData)
    {Back-project correction values to get accurate Output
    volume}
    BackwardProjection(CorrProjData,OutVolData)
  end for
end for
```

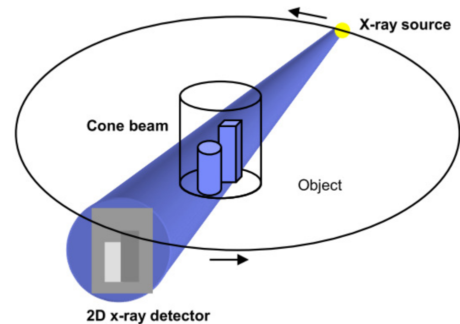


Fig. 4. Illustration of CT geometry in cone beam in 3D space.

Algorithm 3 ForwardProjection

{Arguments: InputProj, OutVolData, CorrProjData}
for i=1 to PROJECTION_HEIGHT **do**
  for j=1 to PROJECTION_WIDTH **do**
    {r_{ij} denotes the ray passing through pixel InputProj[i,j]}
    {Trace the path of the ray through reconstruction grid using Jacob's algorithm [14]}
    raysum = 0, sum_w = 0
    while r_{ij} traverses through the reconstruction grid **do**
      (x,y,z) = current voxel traversed by ray r_{ij}
      w = length of ray within the voxel (x,y,z)
      raysum += w * OutVolData[x,y,z]
      sum_w += w
    end while
    CorrProjData[i,j]=(InputProj[i,j] - raysum)/sum_w
  end for
end for

Algorithm 4 BackwardProjection

{Arguments: CorrProjData, OutVolData}
for k=1 to RECON_DEPTH **do**
  for i=1 to RECON_HEIGHT **do**
    for j=1 to RECON_WIDTH **do**
      {Use projection matrix calibration data (Feldkamp's algorithm [15]) to compute the region of CorrProjData that will map to voxel OutVolData[k,i,j]}
      (u,v) = centroid of the region in CorrProjData that maps to voxel OutVolData[k,i,j]
      {Refine voxel OutVolData[k,i,j] using bi-linearly interpolated correction values around (u,v)}
      OutVolData[k,i,j] += BiLinearInterpolation(CorrProjData,u,v) * CorrectionFactor
    end for
  end for
end for

1) Vectorization: In order to take advantage of the MIC VPUs, we have vectorized the innermost loop of our FIR filter algorithm. For efficient vectorization of the loop, it is preferred that there are no conditional branch statements within the loop [16], [17], [18], [19]. So, we had to remove the 'if' statement present within the innermost loop (Algorithm 1). We have addressed this by sufficiently padding each row of the input image with zero values on either sides. With this change, though the MIC compiler could vectorize the modified code, we did not get much performance benefit from the vectorization because of interleaved data accesses to the real and imaginary values. So, we stripped the real and imaginary values of both image and coefficient data to separate arrays. With this modification, we obtained an average (across all tap sizes) of 26% performance improvement over baseline (Refer Section V-A, Figure 5). The vectorized code snippet is shown in Listing 3.

```
__declspec ( target (mic) )
void ComplexFilterFIR_OptVectorization(
    float *real_src, float *img_src,
    float *real_coeff, float *img_coeff,
    float *dest, int xSize, int ySize,
    int N) {
  const int OFFSET = (N+1)/2;
  const int HN = N/2;
  #pragma omp parallel default (shared)
  {
    #pragma omp for nowait
    for (int k=0; k<ySize; k++) {
      int kInd1 = k*(xSize+N+1);
      int kInd2 = k*(xSize*2);
      for (int j=0; j<xSize; j++) {
        float sumI=0;
        float sumQ=0;
        float I,Q,C,D;
        for (int i=0; i<=N; i++) {
          I=real_src[kInd1+j+OFFSET+i-HN];
          Q=img_src[kInd1+j+OFFSET+i-HN];
          C=real_coeff[N-i];
          D=img_coeff[N-i];
          sumI += I*C-Q*D;
          sumQ += Q*C+I*D;
        } //end for i
        dest[kInd2 + j*2]      =sumI;
        dest[kInd2 + j*2 + 1]  =sumQ;
      } //end for j
    } //end for k
  } //end paralell
} //end ComplexFilterFIR_OptVectorization
```

Listing 3. FIR Filter Parallelized and Vectorized Code Snippet

2) Loop Splitting: The vectorized FIR filter code snippet shown in Listing 3 suffers from repeated loading of coefficient values (both real and imaginary) for every image pixel. We applied an optimization called 'loop splitting' [18], [20] to avoid this repeated loading of coefficient values. As highlighted in Listing 4, the innermost loop is split into two loops: the original loop with index 'i', and the new loop with index 'iOuter'. Please note that the iteration space of the original innermost loop of Listing 3 (highlighted) is shared between the two split loops of Listing 4. Having done this, we could hoist the loading of 16 values of C and D to outside the 'j' loop. This results in a small increase of runtime memory footprint compared to vectorized code. However, due to the reduction in redundant memory loads of co-efficient values, loop splitting results in an average of 53% performance improvement over vectorization (Section IV-A1).

3) Loop Unrolling: Loop unrolling [18], [20] is a loop optimization technique which replicates the body of a loop some number of times called the unrolling factor 'u' and iterates by step 'u' instead of step 1. Unrolling can improve the performance by reducing the loop overheads, increasing instruction parallelism, and improving register, data cache, or TLB locality. We implement loop unrolling in the FIR filter algorithm for higher filter orders, i.e., $N \geq 128$. For these filter orders, the 'iOuter' loop is incremented in steps of 128 and hence the inner most 'i' loop trip count is increased to 128. This allows us to unroll the inner most 'i' loop with an unrolling factor of 8. Listing 5 shows the FIR filter code

```
#pragma omp parallel default (shared)
{
  #pragma omp for nowait
  for (int k=0; k<ySize; k++) {
    int kInd1 = k*(xSize+N+1);
    int kInd2 = k*(xSize*2);
    for(iOuter=0; iOuter<=N; iOuter+=16) {
      float C[16], D[16];
      for (int l=iOuter; l<iOuter+16; l++) {
        C[l-iOuter] = real_coeff[N-l];
        D[l-iOuter] = img_coeff[N-l];
      } //end for l
      for (int j=0; j<xSize; j++) {
        float sumI=0;
        float sumQ=0;
        float I,Q,C,D;
        for (int i=iOuter; i<iOuter+16; i++) {
          I=real_src[kInd1+j+OFFSET+i-HN];
          Q=img_src[kInd1+j+OFFSET+i-HN];
          sumI += I*C[i-iOuter]-Q*D[i-iOuter];
          sumQ += Q*C[i-iOuter]+I*D[i-iOuter];
        } //end for i
        dest[kInd2 + j*2]      +=sumI;
        dest[kInd2 + j*2 + 1]  +=sumQ;
      } //end for j
    } // end for iOuter
  } //end for k
} //end parallel
```

Listing 4. FIR Filter Code Snippet for Loop Splitting Optimization

snippet for the loop unrolling optimization. Loop unrolling showed mixed results.

```
#pragma omp parallel default (shared)
{
  #pragma omp for nowait
  for (int k=0; k<ySize; k++) {
    ...
    for(iOuter=0; iOuter<=N; iOuter+=128) {
      float C[128], D[128];
      for (int l=iOuter; l<iOuter+128; l++) {
        C[l-iOuter] = real_coeff[N-l];
        D[l-iOuter] = img_coeff[N-l];
      } //end for l
      ....
      #pragma unroll(8)
      for (int i=iOuter; i<iOuter+128; i++) {
      ....
```

Listing 5. FIR Filter Code Snippet for Loop Unrolling Optimization

B. SART

The code for both forward and backward projection of SART is parallelized using OpenMP to take advantage of the MIC cores. The parallelization is done at the level of the outermost 'for' loop for both forward and backward projection so that different threads can work in parallel on different projection image rows, thereby maximizing spatial locality. To optimize the data transfer and minimize offload overheads, we concatenated all the projection data and made a single offload call to MIC that would execute the forward and backward projection on all of the projection data iteratively.

The SART forward and backward projection code included many computations within conditional branch statements and hence had to be modified to be amenable for vectorization. The first modification was to split the innermost 'j' loop in Algorithms 3 and 4 into two loops: the original 'j' loop, and the new innermost 'jInner' loop. Please note that in the modified code Listing 6, the 'j' loop is incremented in steps of 16, and the 'jInner' loop trip count is set to 16 (highlighted). The reason behind setting the trip count of the 'jInner' loop to 16 is to keep the MIC VPUs busy, which support 16 SP FP operations. Also, note that the 'jInner' loop is split into three loops (highlighted): the first 'jInner', the second 'jInner' with 'while' and the third 'jInner'; This split was required because compiler can not vectorize a 'while' loop. We also had to apply loop interchange optimization for the second 'jInner' to facilitate vectorization (Listing 6).

We employed a voxel-driven parallelization approach for the backward projection, where each MIC thread updated voxel values of one slice of the reconstruction grid. This ensured a lock-free algorithm as each thread updates an independent set of voxels. This also maximizes the cache locality and minimizes false sharing across threads.

Listing 6 shows the parallelized and vectorized pseudo code for SART forward projection. A similar approach was used for the backward projection.

V. Experimental Results and Analysis

In this section we report the achieved performance on Intel® Xeon Phi™ (KNC) implemented with Intel's heterogeneous offload model. Intel® Xeon Phi™ processor has 60 cores operating at 1.053 GHz. Each core can run upto 4 hardware threads and has 32KB L1 instruction cache and 32KB L1 data cache. Each core also has 512 KB L2 cache. The device has 8 GB of GDDR5 RAM.

We use Intel compiler version 13.1 using -O3 -openmp switches. We use latest MPSS stack. We have used dynamic scheduling of threads in OpenMP resulting in improved performance. All performance numbers are average of 3 runs.

A. Performance Analysis of FIR

Figure 5 plots the FIR compute times when the different optimization strategies discussed in Section IV-A are incrementally applied. Though MIC supports up to four threads per core, we could get the best performance by launching just two threads per core. This may be because FIR code is able to saturate functional units in MIC using two threads. As noted before, vectorization yields an average of 26% performance improvement over the baseline code. Adding loop splitting further boosts the performance by 53% on an average by optimizing locality and memory accesses. Unrolling does not yield a significant benefit.

To understand the efficiency of our FIR implementation on KNC, we compare the achieved compute and memory throughput with the theoretical peak throughputs. The theoretical peak single precision compute throughput and memory bandwidth offered by KNC are provided in Table I. This

```
Vectorized_Forward_Projection(float *InProjData,
    float *OutVolData, float *CorrProjData) {
  #pragma omp parallel default (shared) {
    float ray[16], ln[16], sum_w[16];
    unsigned short FLAG;
    #pragma omp for nowait
    for (int k=0; k<PROJ_H; k++) {
      for (int j=0; j<PROJ_W; j+=16) {
        for (int jInner=j; jInner<j+16; jInner++) {
          1. For the pixel InProjData[k,jInner],
             compute the ray r through
             InProjData[k,jInner]
            and assign it to ray[jInner-j].
        } //end for jInner
        2. Initialize FLAG, ln[], and sum_w[] to 0.
        while (FLAG < 0xFFFF) {
          for (int i=0;i<16;i++) {
            if (ray[i] passes through the output
                volume) {
              3. Compute the current voxel (x,y,z)
                 of OutVolData through which
                 ray[i] passes.
              4. Compute the intersection length
                 'w' of ray[i] through the voxel
                 (x,y,z).
              5. ln[i] += w * OutVolData[x,y,z]
              6. sum_w[i] += w
            } //end if
            else { // Set i'th bit of FLAG to
                indicate completed traversal for
                ray i
              7. FLAG = FLAG | (1<<i);
            } //end else
          } //end for i
        } //end while
        for (int jInner=j; jInner<j+16; jInner++) {
          8. CorrProjData[k,jInner] =
               (InProjData[k,jInner]-ln[jInner-j])
            /sum_w[jInner-j]
        } //end for jInner
      } //end for j
    } //end for k
  } //end parallel
} //Vectorized_Forward_Projection
```

Listing 6. Parallelized and Vectorized Pseudo Code Snippet for SART Forward Projection

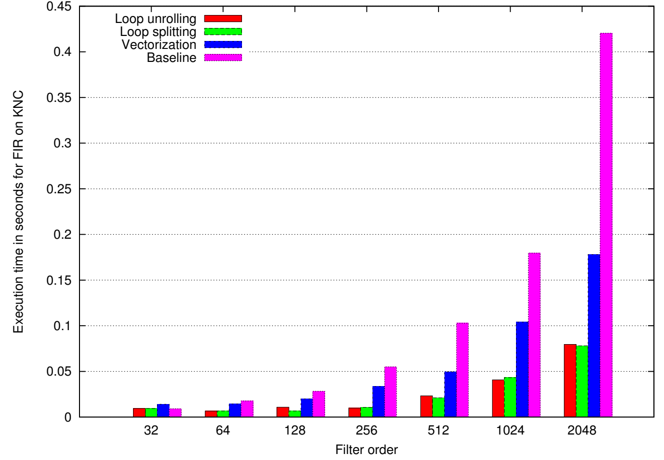


Fig. 5. FIR execution times (for 4096 x 512 image) in seconds on Intel® Xeon Phi™ (KNC)

Metric	KNC
Theoretical SP GFlop/s	2021
Theoretical GB/s	320
Achievable SP GFlop/s	1729
Achievable GB/s	159

TABLE I
THEORETICAL AND ACHIEVABLE PEAK SINGLE PRECISION COMPUTE AND MEMORY THROUGHPUTS FOR KNC [21].

table also shows the achievable peak single precision compute throughput for SGEMM benchmark and achievable peak memory bandwidth for STREAM benchmark [21].

We estimate the total number of floating point operations and bytes read/written in the FIR algorithm (caching effects are ignored) and calculate the achieved compute and memory throughput based on the actual execution time (Table II). We notice that the FIR algorithm performs approximately twice the number of memory operations than compute (8 GFLOP compute and 17.2 GB memory accessed). Since the theoretical single precision compute throughput of the KNC platform is approximately 6x higher than the memory bandwidth offered (Table I), we see that the overall time is largely dominated by the memory accesses. Hence, we are not able to saturate the cores and we operate at only 17% of the theoretical peak compute throughput. FIR performs a streaming access to the data and we see that we are able to utilize the cache effectively through the loop splitting optimization and avoiding interleaved access to real and imaginary data values (the achieved memory performance is more than twice the peak memory throughput due to caching effects).

To confirm our analysis, we profiled our baseline FIR and optimized FIR (with all three optimizations) code. Baseline FIR has L1 cache hit ratio 0.98, vectorization efficiency 50% and CPI (Clocks per instruction) 3.47. With optimized code, L1 cache hit ratio increases to 1.00, vectorization efficiency increases to 100% and CPI improves to 2.04. We observed that our optimizations improved cache utilization and vectorization and as a result the CPI approaches ideal CPI of 2.0.

B. Performance Analysis of SART

Table III presents the execution time for SART to process 180 projections (half scan) for three different reconstruction grid resolutions (128x128x128, 256x256x256 and 512x512x512). Each projection image is of resolution 512x512. In case of SART, we could get the best performance by launching three threads per core.

The SART algorithm involves projecting the current ap-

Metric	Algorithm Characteristics	Achieved throughput
FIR compute SP	8 GFLOP	345 GFLOP/s
FIR memory	17.2 GB	740 GB/s

TABLE II
FIR ALGORITHM COMPUTE AND MEMORY CHARACTERISTICS AND ACHIEVED SINGLE PRECISION COMPUTE AND MEMORY THROUGHPUTS ON KNC. INPUT SIZE: 4096 X 512 AND FILTER TAPS = 512.

Grid size	Num. of Projections	Total Execution Time	Forward Projection Execution time	Backward Projection Execution time
512^3	180	13.13	8.88	4.25
256^3	180	3.55	2.45	0.9
128^3	180	1.67	1.18	0.49

TABLE III
SART EXECUTION TIME IN SECONDS ON INTEL® XEON PHI™ (KNC).

Algorithm	Compute Characteristics (GFLOP)	Memory Characteristics (GB)
ForwardProjection	420	112
BackwardProjection	930	388

TABLE IV
COMPUTE AND MEMORY CHARACTERISTICS OF SART FORWARD AND BACKWARD PROJECTION PHASES FOR 512×512 PROJECTED IMAGE SIZE AND 512^3 RECONSTRUCTION VOLUME

proximate reconstruction space using forward projection, then the errors are backprojected using backward projection, thus deriving better approximation. This process repeats for all available projections. Hence to evaluate SART, we analyze the performance of ForwardProjection and BackwardProjection algorithms.

Similar to the FIR analysis, we estimate the compute and memory characteristics of the SART forward and backward projection phases for a 512x512x512 reconstruction. We ignore the caching effects while estimating the GBytes accessed. Table IV presents both the compute and memory characteristics of the forward and back projection phases for one iteration. The operations in SART are also of single precision. Contrary to FIR, for the SART algorithm, compute to memory access ratio is approximately 4 for forward projection; for backward projection, this ratio is approximately 2.4. However, the theoretical and achievable peak compute to memory throughput ratios are 6x and 10x respectively (refer to Table I). Therefore, we conclude that the SART algorithm is also dominated by the memory access time.

Table V presents the achieved SART forward and backward projection compute and memory throughput based on the characteristics of algorithms (Table IV) and the actual execution times (Table III). We observe that the forward projection phase has very poor memory and computational efficiency. This is due to non-contiguous memory accesses of the reconstruction grid voxels during the ray traversal. Closely accessed voxels would have good locality in 3D. However, in 1D these accesses translate into non-contiguous access to memory resulting in

Algorithm	Achieved compute throughput (GFLOP/s)	Achieved memory throughput (GB/s)
ForwardProjection	47.3	12.63
BackwardProjection	219	91.3

TABLE V
ACHIEVED COMPUTE AND MEMORY THROUGHPUT FOR SART FORWARD AND BACKWARD PROJECTION PHASES ON KNC FOR 512×512 PROJECTED IMAGE SIZE AND 512^3 RECONSTRUCTION VOLUME

poor cache and memory performance. Since the total time is dominated by the memory access time, this also brings down the achieved compute throughput.

The backward projection phase reports better achieved memory throughput as compared to the forward projection phase, i.e., about 50% of the achievable memory bandwidth as reported in Table I. This is because accesses to the reconstruction grid voxels are optimized for spatial locality. However, accesses to the correction data to perform a bilinear interpolation of correction values around the centroid to update the voxel values are localized in 2D, which translates to non-contiguous memory accesses. This may be the reason for drop in memory access performance.

To confirm our analysis, we profiled the optimized forward and backward projection code. Forward projection has L1 hit ratio 0.927, vectorization efficiency 56% and CPI 7.35. Backward projection has L1 hit ratio 0.99, vectorization efficiency 68% and CPI 4.14. Lower L1 hit ratio and higher CPI for forward projection versus backward projection is due to non-contiguous memory accesses in 3D versus 2D. We believe that, there is room for performance improvement in SART and needs further analysis.

To summarize, our analysis with SART indicates that algorithms that have irregular memory accesses or non-contiguous accesses may not be able to saturate the compute capacity of KNC. Non-contiguous memory accesses are performed using gather/scatter instructions in MIC, however performance of the same is limited by the actual load latencies. Our analysis of FIR shows that it has a regular access pattern with 1D spatial locality and hence achieves better performance on MIC. In addition to memory and cache performance, the scope for vectorization offered by the algorithm also plays an important role in determining the achievable performance on KNC.

As MIC is based on x86, it provides a natural extension to conventional x86 programming models, like OpenMP. We found that MIC programming has an easy learning curve. With a knowledge of OpenMP, basic porting of a program to execute on MIC can be achieved very quickly. Furthermore, the MIC programming model shields the programmer from the job of handling device memory aspects such as allocation/deallocation, data transfer to/from device memory, etc. Also, clauses such as 'alloc_if', 'free_if' and 'no_copy' are provided to optimize data transfers. Performance optimization on the MIC platform, on the other hand, is a more involved activity and requires insight into compiler optimization techniques, cache behaviour, access patterns etc. However, these concepts are inherent principles of parallel programming even on multicore processors and can be extended to MIC optimization seamlessly.

VI. CONCLUSION

Accelerators such as Intel® Many Integrated Core (MIC) are enablers of high performance, high throughput and power efficient computing. However, efficient utilization of its compute capability requires expressing underlying data-parallelism in the problem. MIC architecture allows more traditional

approach for expressing parallelism in programs with use of OpenMP directives. Programmers rely on auto-vectorization features of compilers to generate code for wide vector processing units (VPU) on MIC co-processor.

This paper presents our experiences using the MIC platform. We have evaluated its performance in the context of medical imaging and signal processing using two algorithms: complex Finite Impulse Response filter (FIR) used in ultrasound image processing and Simultaneous Algebraic Reconstruction Technique (SART) used in computed tomography reconstruction. Our evaluations indicate that the MIC co-processor holds a lot of promise to be an easy-to-program accelerator of the future that also demonstrates good performance. However, extracting the best performance depends on efficient vectorization of loops and cache utilization.

REFERENCES

[1] Maurice Herlihy, "The multicore revolution: the challenges for theory," in *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, Berlin, Heidelberg, 2007, pp. 1–8, Springer-Verlag.

[2] John Shalf, "The new landscape of parallel computer architecture," *Journal of Physics: Conference Series*, vol. 78, no. 1, pp. 012066, 2007.

[3] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli, "State-of-the-art in heterogeneous computing," *Sci. Program.*, vol. 18, pp. 1–33, January 2010.

[4] "Intel xeon phi coprocessor system software developers guide," http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide.

[5] "The openmp api specification for parallel programming," http://openmp.org/wp/.

[6] "The message passing interface (mpi) standard," http://www.mcs.anl.gov/research/projects/mpi/.

[7] "Intel xeon phi coprocessor developer's quick start guide," http://software.intel.com/sites/default/files/article/335818/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf.

[8] L.J. Morales-Mendoza, Y. Shmaliy, E. Morales-Mendoza, and R. Ortega-Almanza, "P-step unbiased fir filter to the ultrasound image processing," in *Electronics, Communications and Computer (CONIELECOMP), 2010 20th International Conference on*, 2010, pp. 96–101.

[9] Luis Javier Morales-Mendoza, Ren Fabin Vzquez-Bautista, Efrn Morales-Mendoza, and Yuriy Semenovich Shmaliy, "Speckle noise reduction in ultrasound imaging using the key points in low degree unbiased fir filters," *Computacin y Sistemas*, vol. 16, pp. 287–295, 2012.

[10] A H Andersen and A C Kak, "Simultaneous algebraic reconstruction technique (sart): a superior implementation of the art algorithm.," *Ultrasonic Imaging*, vol. 6, no. 1, pp. 81–94, 1984.

[11] Benjamin Keck, Hannes Hofmann, Holger Scherl, Markus Kowarschik, and Joachim Hornegger, "GPU-accelerated SART reconstruction using the CUDA programming environment," in *Proceedings of SPIE*, Lake Buena Vista, 2009, vol. 7258.

[12] Hengyong Yu and Ge Wang, "Sart-type image reconstruction from a limited number of projections with the sparsity constraint," *International Journal of Biomedical Imaging*, vol. 2010, 2010.

[13] "Fir filter algorithm implementation using intel sse instructions," http://download.intel.com/design/intarch/papers/323411.pdf.

[14] Filip Jacobs, Erik Sundermann, B De Sutter, Mark Christiaens, and Ignace Lemahieu, "A fast algorithm to calculate the exact radiological path through a pixel or voxel space," *Image Processing*, vol. 6, no. 1, pp. 8994, 1998.

Disclaimer: Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

[15] "L. a. feldkamp, l. c. davis, and j. w. kress, practical cone-beam algorithm, j. opt. soc. amer. a1(6), pp. 612619," 1984.

[16] "Intel sse4 programming reference," http://software.intel.com.

[17] "Intel advanced vector extensions programming reference," http://software.intel.com.

[18] Steven S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[19] Ken Kennedy and John R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[20] David F. Bacon, Susan L. Graham, and Oliver J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[21] "Intel xeon phi product family performance," http://www.intel.in/content/dam/www/public/us/en/documents/performance-briefs/xeon-phi-product-family-performance-brief.pdf.