Efficient FDK Algorithms on SIMD-accelerated Processors

Peng Chen[1,a)] Mohamed Wahib[1,4] Shinichiro Takizawa[2] Takahiro Hirofuchi[2]
Hirotaka Ogawa[2] Satoshi Matsuoka[3,4]

Abstract: Computed Tomography (CT) is a widely used 3D imaging technology that requires compute-intense algorithms to generate volume data (or images). We propose a collection of novel back-projection algorithms that reduce the arithmetic computation and improve data locality. We also implement novel algorithms as efficient back-projection kernels that are performance portable over a wide range of CPUs. Unlike the conventional approaches that use OpenMP and target-specific SIMD intrinsics, we employ a high-level OpenCL implementation to generate the vectorized code and use the OpenCL local memory to prefetch the pixels at sub-pixel precision in a regular memory access fashion. Performance evaluation using a variety of Intel CPUs generations demonstrates that our back-projection implementation runs up to 10 times faster than the multi-threading optimized implementation.

Keywords: FDK, OpenMP, SIMD, OpenCL

1. Introduction

Computed Tomography (CT) is a useful 3D imaging technique in several fields such as medical diagnosis, non-destructive inspection, and scientific analysis. As investigated in literature [1], the FDK [*1] algorithm is widely regarded as the standard methodology to convert a set of projection images to volume data (or 3D image). There exist two intensive computations in FDK, namely filtering computation and back-projection, their computational complexities are as high as $O(N^2 log(N))$ and $O(N^4)$, respectively. Hence, the back-projection is often the computational bottleneck of the FDK algorithm. To meet the critical demands for rapid image reconstruction, several kinds of accelerators are employed to improve the computational performance of back-projection such as Application Specific Integrated Circuits [3], Field-Programming Gate Array (FPGA) [4], [5], Digital Signal Processor (DSP) [6], Intel Xeon-Phi accelerator [7], and Graphics Processing Unit (GPU) [8], [9], [10], [11]. In this paper, we focus on optimizing back-projection on Intel multicore CPUs by algorithm innovations and parallel computing techniques.

There are strong motivations to optimize the FDK algorithm on Intel ×86 CPUs [12]. Firstly, integrating extra-accelerators into CT systems increases the system complexity, as well as lead to considerable costs, e.g. hardware and software development. The use of Intel ×86 processors would be preferable. It is cost-less and also energy-efficient to access these kinds of powerful multicore processors since they are a basic component in most embedded devices, laptops, workstations, and cloud servers. Secondly, it is very attractive to run OpenCL-optimized kernels on SIMD processors to meet the critical demand for high performance and avoid CPU-accelerator bandwidth bottleneck for high throughput. As presented in [13], the adoption of more vector units in modern CPUs narrows the performance gap between CPUs and GPUs (a heterogeneous accelerator as in [14]). The only use of CPUs with tuned algorithms can meet the critical timing requirement in several applications such as image reconstruction in this paper. Finally, as an open standard for parallel programming, OpenCL expresses parallelism to ultimately utilize the computing resources as a combination of Single Instruction Multiple Thread (SIMT) and Single Instruction Multiple Data (SIMD) programming model [15], [16]. Prior work in [17] demonstrates the performance advance of OpenCL in comparison to OpenMP and Intel Threading Building Blocks (TBB) on multicore CPUs. More importantly, the local memory available in OpenCL is highly optimized by the cache [17]. On one hand, it can be employed as the user-managed cache for reducing the host memory access to some extent; On the other hand, it also can be used to perform the efficient intra-group communication between work-items.

There are several challenges to optimize FDK algorithms on Intel CPUs with SIMD architecture. Firstly, back-projection is a well-researched algorithm and its optimization often requires acceleration techniques of low-level languages, e.g. paralleling algorithms described with OpenCL, SIMD instructions on time-consuming kernels [18]. Secondly, back-projection is often a computational bottleneck. It is essential to improve its performance by reducing the arithmetic computation and saving memory access. Thirdly, data locality and cache-friendly code are critical to improving the computation efficiency of back-projection

[1] AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory, National Institute of Advanced Industrial Science and Technology
[2] National Institute of Advanced Industrial Science and Technology
[3] Tokyo Institute of Technology
[4] RIKEN Center for Computational Science
[a)] chin.hou@aist.go.jp
[*1] A convolution-back-projection methodology (also known as Filtered Back-Projection, FBP) was innovated by Feldkamp, Davis, and Kress [2] (called FDK) for CT image reconstruction in 1984.

kernels. To better utilize the rich cache resource [19], a data blocking mechanism is required; to meet the requirement of contiguous memory access pattern and minimize the cache miss frequency, data layout rearrangement is also important. Fourthly, the theoretical peak performance of CPU is boosted by SIMD intrinsic on vector units. Vectorization and optimizing instruction pipelines for back-projection inner-kernel helps to get closer to the chip's peak performance.

We optimize the back-projection for FDK algorithms on Intel multicore CPUs using OpenCL. The OpenCL kernel on CPU platform can directly access the host memory, and thus benefits to reduce the overheads of memory allocation and data movement between host and devices. We improve the data locality by rescheduling the loop order and transposing the projection images and volume data to ensure continuous memory access (or a regular memory access pattern), unlike the work in [18] relies on gather load intrinsic. The evaluated result has proved the validity of our methodology using a variant of Intel CPUs. A novel sub-line algorithm is also proposed for an efficient interpolation at sub-pixel precision. Note that single-precision was used in all computations. Based on this algorithm, the memory access is kept to be contiguous, the sub-line interpolation is blocked via local memory to be performed with high cache hits and accelerated with automated vectorization. Finally, the performance evaluation on several generations of Intel CPUs, e.g. Xeon E5-2630 v4, Core i7-9700K, shows that our back-projection kernel performs up to 10× faster than the multi-threading optimized baseline implementation. The contributions in this paper are three-folds:

- We propose a collection of novel back-projection algorithms that reduce the computational cost of projection operations and improve data locality.
- We implement the first Intel CPU-specified back-projection kernel by OpenCL.
- We demonstrate that our OpenCL kernels achieve outstanding performance as up to 10× faster than the multi-threading optimized implementations on a variant of Intel CPUs.

The rest of this paper is organized as follows: In Section 2, we introduce the background. Section 3 illustrates the proposed algorithms. Section 4 shows the evaluated result. In Section 5, we elaborate on the related work. Finally, Section 6 concludes.

2. Background

In this section, we briefly introduce the details of the FDK algorithm and describe the basics of Intel OpenCL SDK for multicore CPUs.

2.1 FDK algorithm

This section illustrates the 3D image reconstruction algorithm for Cone-Beam Computed Tomography (CBCT) as presented by Feldkamp et al. [2] including CBCT geometry and back-projection algorithm. Note that readers can refer to [2], [20] for more details of the filtering computation, which is an important computing step.

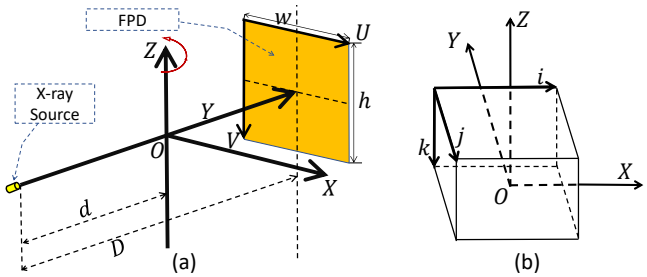


Fig. 1: The triangular geometry of CT system.

Listing 1: The back-projection baseline implementation (as in [22]) that is optimized by OpenMP as the pragma syntax in line 4. nProj is the number of projections. The sizes of the 2D projections, volume data and projection matrix are heigh×width, nz×nx×ny, and 3×4, respectively.

```
1  void bpBaseline_mt(float img[nProj][height][width], float
     mat[nProj][3][4], float volume[nz][ny][nx])
2  {
3      for (int s = 0; s < nProj; s++){
4        #pragma omp parallel for
5        for (int k = 0; k < nz; k++)\{
6          for (int j = 0; j < ny; j++){
7            for (int i = 0; i < nx; i++){
8              float vec[4] = {i,j,k,1.f};//coordinate
9              float z = dot4(mat[s][2], vec); //dot
10             float f = 1.f/z;
11             float x = dot4(mat[s][0], vec)*f; //dot
12             float y = dot4(mat[s][1], vec)*f; //dot
13             float val = subPiexl(img[s], x, y);
14             float weight = f*f;  //compute weight
15             volume[k][j][i] += val*weight;//update
16     }  } } } // s, k, j ,i
17 }
```

2.1.1 Geometry of CT system

The Figure 1 shows a triangular geometry of CBCT system. The X-ray source is a kind of microfocus x-ray tube, the Flat Panel Detector (FPD) is a class of digital radiography imaging sensor like digital photography. The distances of the source to the rotation axis (the Z-axis) and FPD are d and D respectively. The sizes of of FPD in a unit of pixel are **width** and **height** (as *w* and *h* in Figure 1a), respectively. Note that the U-axis and V-axis of FPD are parallel to X-axis and Z-axis, respectively. As figure 1b shown, the size of 3D volume data in a unit of voxel [*2]are **nx**, **ny**, and **nz**, respectively. The default data layout of a volume data is row-major order as the direction of *i* in Figure 1b. As a typical pinhole geometry [21], all geometric information can be presented as a matrix of size 3×4 (called projection matrix), which are used for back-projection computation, e.g. projecting a voxel to the plane of FPD as the variable of *mat* in Listing 1. For simplification we ignore the computation of obtaining the projection matrix via the geometry information, the detailed formulation is elaborated in [22], [23]. In Algorithm 1, the operation of mapping a 3D point (i, j, k) to the plane of FPD is called projection computation as the runnable codes of lines 8∼12.

2.1.2 Back-projection

As Listing 1 shown, we directly present the back-projection algorithm as implemented in RTK library by Simon et al. [22]. The

[*2] A voxel is named as a point in volume data.

Listing 2: The customized function for inner production. Note that v1[3] is a constant of 1.0 as in line 8 of Listing 1.

```
1 float dot4(float v0[4], float v1[4])
2 {
3     return v0[0]*v1[0]+v0[1]*v1[1]+v0[2]*v1[2]+v0[3];
4 }
```

Listing 3: The subPixel function for interpolation at sub-pixel precision.

```
1  float subPiexl(float img[height][width], float x, float y
     )
2  {
3     int nx = (int)x; //convert to integer
4     int ny = (int)y; //convert to integer
5     float dx = x - (float)nx; //get sub-pixel coordinate
6     float dy = y - (float)ny; //get sub-pixel coordinate
7     //horizontal interpolation
8     float s0=img[ny][nx]*(1.f-dx)+img[ny][nx+1]*dx;
9     float s1=img[ny+1][nx]*(1.f-dx)+img[ny+1][nx+1]*dx;
10    //vertical interpolation
11    float val = s0*(1.f-dy) + s1*dy;
12    return val;
13 }
```

multi-threading via OpenMP [24] (as the syntax in line 4) is employed to speed up the computation by taking advantage of the multi-cores of the processor. As an inputted argument, the **img** is the filtered projection, namely the output data of the filtering computation, its dimensions may be expressed as (nProj, height, width). The **mat**, projection matrix as illustrated in the prior paragraph, is used to project the voxel to the FPD plane. The projected coordinate is written as x and y in lines 11 and 12. The value of z (line 9) is used to derive the projection coordinate, as well as a weighting factor to update the volume data in lines 14∼15. The **dot4**, a customized function for the inner product as in Listing 2, was called three times in the most inner loop. As listing 3 shown, an implementation of bilinear interpolation function, namely **subPixel**, is used to fetch the density value of inputted 2D image. To obtain a single value at sub-pixel precision, four different values are loaded and 18 arithmetic operations are performed. It is clear that the computational complexity of back-projection is $O(N^4)$.

2.2 Intel OpenCL SDK for Multicore CPUs

The OpenCL abstracts an open standard platform for general-purpose parallel programming and provides a uniform programming API (Application Programming Interface) that is often used to write portable and efficient code for a diverse of accelerators, e.g. CPUs, GPUs, and FPGAs. As one of the open and portable standards defined by Khronos Group, OpenCL is used to develop software targeting parallel computing platforms using a host/device programming model. Programming by OpenCL involves writing a host code, often in C/C++, that executes as the host, and a kernel code developed in C, that runs on the device or accelerator. The host codes are often compiled by a general C/C++ compiler (e.g. gcc/g++), and the kernel codes are built at run-time for the specified processors. Also, OpenCL provides APIs to manage the accelerators and control the data movement between host and accelerators. The computing unit in OpenCL is a work-item. Work-groups is a collection of work-items and abstracted as a multidimensional descriptor called an NDRange. The work-items within the same work-group can exchange data using the local memory.

Intel OpenCL SDK for Multicore CPUs allows developers to take advantage of the SIMD-accelerated multicores using SIMT (Single Instruction Multiple Threads) programming model and high-level compiler to automate the vectorized binaries, that is completely different from the prevalent optimization technique using both SIMD vectorization and multithreading such as OpenMP, Intel TBB [25] and pthread library [26]. Note that OpenCL compilers employ an even wider variety assembler transformations beyond the classic ×86 ecosystem as compilation optimization, resulting in the OpenCL is not performance portable. Importantly, the access to local and constant memory can be highly optimized by the cache, their sizes are 32KB and 128KB, respectively. We can use these memories to optimize applications effectively. We just briefly introduce the concept of OpenCL, more details can be found in [27].

2.3 Terminology

We define the image reconstruction problem and performance metrics in this section.

$width \times height \times nProj \rightarrow nx \times ny \times nz$ is defined as the image reconstruction **problem**, where $width \times height$ indicates the size of input projections and $nx \times ny \times nz$ is the size of output volume. The performance metric of problem maybe written as $\frac{nx*ny*nz}{T*10^9}$, where T denotes the run-time in an unit of second. The performance unit of a kernel is **GUPS**, which means giga-updates per second. Also, we define the ratio of input and output problem as $\frac{\sqrt{width*height}}{\sqrt[3]{nx*ny*nz}}$. Same to the computation of GUPS, the ratio of input and output problem is also independent of the value of nProj in all problems.

3. Proposed Algorithm

This section presents the proposed algorithm. The back-projection is optimized by several methods such as multi-threads (e.g. using OpenMP) and OpenCL. We improve the data locality and memory access pattern by transposing projections and volume data. We also take advantage of OpenCL to speed up the back-projection. To reduce the global memory (or host memory) access, we proposed a novel sub-line algorithm to cache the interpolated data at sub-pixel precision by local memory.

3.1 OpenMP-optimized back-projection

This section describes optimizing back-projection using multi-threads and vectorization. As shown in Listing 1, the compute-intensive back-projection is driven by voxels, which are updated by their mapped pixels at projections. Vectorizing operations and cache-friend memory access can dramatically improve computational performance. We illustrate the detailed optimization step by step as in Listing 4, Listing 5, and Listing 6. Note that all improved variables in Listings are highlighted by gray color.

3.1.1 Improving memory access pattern

This section discusses the memory access pattern. As shown

Listing 4: The proposed back-projection by transposing projections and volume data.

```
1  void bpTranspose_mt(float img[nProj][width][height],
2      float mat[nProj][3][4], float volume[nx][ny][nz])
3  {
4     for (int s = 0; s < nProj; s++){
5        #pragma omp parallel for
6        for (int i = 0; i < nx; i++){
7           for (int j = 0; j < ny; j++){
8              for (int k = 0; k < nz; k++){
9                 float vec[4] = {i,j,k,1.f}; //coordinate
10                float z = dot4(mat[s][2], vec); //dot
11                float f = 1.f/z;
12                float x = dot4(mat[s][0], vec)*f; //dot
13                float y = dot4(mat[s][1], vec)*f; //dot
14                float val = subPiexl(img[s], y, x);
15                float weight = f*f; //compute weight
16                volume[i][j][k] += val*weight; //update
17    } } } } // k, j, i, s
18 }
```

Listing 5: The proposed back-projection by sharing variables. The arguments are same to Listing 4.

```
1  void bpShare_mt(......){
2     //codes & loops for s, i, j are same to line 3~6 of Lising 4
3     ...... ...... ...... ...... ......
4             float vec[4] = {i,j,0,1.f};//coordinate
5             float z = dot4(mat[s][2], vec);//dot
6             float f = 1.f/z;
7             float weight = f*f; //compute weight
8             float x = dot4(mat[s][0], vec)*f;//dot
9             for (int k = 0; k < nz; k++){
10               vec[3] = k; //update k
11               float y = dot4(mat[s][1], vec)*f; //dot
12               float val = subPiexl(img[s], y, x);
13               volume[i][j][k] += val*weight; //update
14    } } } } // k, j, i, s
15 }
```

Listing 6: The proposed back-projection using geometry symmetry. The arguments are same to Listing 4.

```
1  void bpSymmetry_mt(......){
2     //codes & loops for s, i, j are same to line 2~8 of Lising 5
3     ...... ...... ...... //Note: including lines 4~8 of Listing 5
4             for (int k = 0; k < nz/2; k++){
5                vec[3] = k; //update k
6                float y = dot4(mat[s][1], vec)*f; //dot
7                float val = subPiexl(img[s], y, x);
8                volume[i][j][k] += val*weight;//update
9                //geometry symmetry
10               y = height - 1.f - y;
11               val = subPiexl(img[s], _y, x);
12               volume[i][j][nz-1-k] += val*weight;//update
13    } } } } // k, j, i, s
14 }
```

in the Listing 4, we present the basic optimization using transposed projections and volume data. More specifically, we optimize the data access pattern by transposing the two-dimensional projections as seen the argument of *img*. Note that the transposing operation is light-weight and can be performed immediately after filtering computation (out of scope to discuss this operation). Hence, the size of each projection becomes (width, height) as in line 1 of Listing 4. We also reorganize the loop by moving the first loop to the most inner one as in line 7 of Listing 4.

Listing 7: The proposed back-projection by reorganizing loops and using scratchpad memory. The arguments are same to Listing 4.

```
1  void bpScratchpad_mt(......)
2  {
3     #pragma omp parallel for
4     for (int i = 0; i < nx; i++){
5        for (int j = 0; j < ny; j++){
6           float F[nProj], Weight[nProj], X[nProj];
7           float vec[4] = {i,j,0,1.f};
8           for (int s=0; s<nProj; s++){
9              F[s] = 1.f/dot4(mat[s][2], vec);//dot
10             Weight[s] = F[s]*F[s];
11             X[s] = dot4(mat[s][0], vec)*f; //dot
12          }
13          for (int k = 0; k < nz/2; k++){
14             vec[3] = k; //update k
15             float sum=0, _sum=0;
16             float y;
17             for (int s=0; s<nProj; s++){
18                y = dot4(mat[s][1], vec)*f;//dot
19                float val = subPiexl(img[s], y, X[s]);
20                sum += val*Weight[s];//accumulate
21                //geometry symmetry
22                y = height - 1.f - y;
23                val = subPiexl(img[s], y, X[s]);
24                _sum += val*Weight[s];//accumulate
25             }//s
26             volume[i][j][k] += sum; //update
27             volume<[i][j][nz-1-k] += _sum;//update
28    } } } // i, j, k
29 }
```

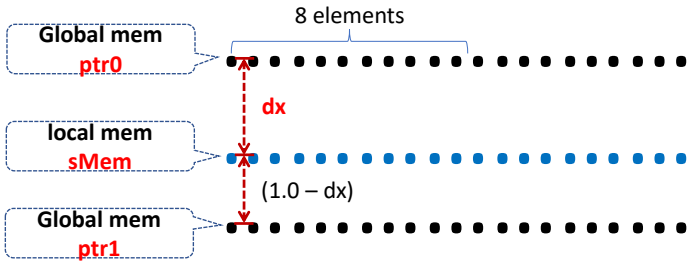
This operation allows us to compute the projections and update the volume data along the vertical direction (or Z direction as in Figure 1). It is because all projections of voxels along k direction (as in Figure 1b) are in a vertical line of FPD plane and thus, the transposed projections contribute to contiguous memory access while fetching pixel intensity using the bilinear interpolation function as shown in Listing 3. More specifically, we can perform row-wise data accesses to projection and volume data by such a transpose operation. More importantly, transpose operation builds a solid foundation for further optimizations as in the following sections.

3.1.2 Reducing arithmetic computation

This paragraph explains reducing arithmetic computation of back-projection. The Listing 5 reduces the computations of listing 4 by sharing data (see the highlighted variables of f, weight, and x in Listing 5) and Listing 6 improves the computation of Listing 5 using characteristics of geometric symmetry (see the highlighted variables of nz/2 and y in Listing 5).

In Listing 5, the values of z and x are constant values when i, j, and projection matrix (supposing the projection angle is θ) are fixed values. It is because of the values of z equal $d-xcos\theta-ycin\theta$ (as introduced in [20], [28]), where x, y are coordinates of voxels that correspond to the indexes of i and j. The values of x are also independent of i and j. It is due to the projections of voxels are parallel to the V-axis (or Z-axis in Figure 1) at the FPD plane as introduced in the earlier paragraph.

As Listing 6 shows, we also take advantage of the geometric symmetry characteristics in the CT system to reduce the arithmetic computations as proposed by Zhao et al. [10]. As shown in line 4, only half of the projection computations for y (line 6) are performed, the symmetric positions of y can be derived in line



(a) The ptr0 and ptr1 are the pointers of global memory for two neighbouring rows, respectively. sMem is the pointer of local memory for caching the sub-line values. the dx is the distance at sub-pixel precision (0≤dx≤1)

```
1 int offset = 0
2 //load elements from with offset
3 float8 v0 = vload8(ptr0, offset);
4 float8 v1 = vload8(ptr1, offset);
5 //linear interpolation
6 float8 v = mix(v0, v1, dx);
7 //store result to local memory
8 vstore8(v, offset, sMem);
```

(b) OpenCL implementation for caching 8 elements. vload8 and vstore8 are built-in intrinsics for meomory access. mix is a linear blend intrinsic.

Fig. 2: Caching linear sub-line of image by local memory and vector operations.

10 as simple as a single instruction rather than using the complex dot4 function (as in Listing 2).

3.1.3 Reducing host memory access

We present the details of reducing host memory access for improving back-projection performance in this section. As shown in Listing 7, we implement an efficient back-projection kernel that moves the iterations for all projections (as the loop for nProj in Listing 1, Listing 4, Listing 5, and Listing 6) to the most inner loop as in line 17. The voxel values are accumulated locally via register files (as declared in line 15) for multiple projections in lines 17~25 and updated to volume data as a single store operation in lines 26~27. In line 6, we use three arrays (namely F, Weight, and X) to cache the reusable values (see f, weight, and x in Listing 5 and Listing 6). The initialization of these variables can be found in line 8~12 and the use of them is shown in lines 19~24 (see the highlighted variables). The cost of using these variables is lightweight due to their small sizes and cache-friend access patterns (as loops in line 13 and line 17). In other words, the contiguous accesses to these variables are restricted in the buffers with very limited sizes. Note that we introduce the details on employing local memory to cache them in later sections.

3.1.4 Vectorizating operations

In this section, we present the details of employing vector units to tune the projection operation and interpolation computation in back-projection.

Regarding the projection computation (e.g. the computation for x, y, and z in Listing 1), each inner product is performed on two vectors of size 1×4 at single precision, such an operation can be perfectly performed by the vector unit, e.g. the built-in function *dot* is employed for this computation in our OpenCL-based implementation (as will be elaborated later). Note that we benefit from the well-aligned projection matrix of size 3×4 in back-projection algorithms. In [28], the projection computation is not aligned for vector units due to unorganized data access and computations.

Due to the uses of transposed projections and volume data, the linear interpolation operation can perform data access in a regular pattern and its computation can also be well vectorized with the improved data locality. Such an optimization can be applied in all proposed algorithms as in Listing 4~7. As Figure 2a shown, the bilinear interpolations focus on two rows of projections and the data is also can be accessed in a contiguous pattern including reading data from projections and writing data to volume data.

3.2 OpenCL-optimized back-projection

In this section, we take advantage of OpenCL to implemented a collection of back-projection kernels according to the techniques used in the previous section, e.g. reducing computation by sharing data and geometry symmetry, reducing host memory access by batched processing. We also improve the interpolation operation by local memory. As an illustrative example shown in Listing 8, we implement a back-projection kernel corresponding to Listing 7. we explain the used techniques in this kernel (namely Listing 8) as follows:

(i) As the argument *mat* shown in kernel of *symmetry_lm_cl*, we employ constant memory to cache those data. the size of each projection matrix is as small as 48B (sizeof(float)*4*3) and greatly smaller than the capacity of constant memory of size 128KB (as introduced in Section 2.2).

(ii) Regarding the index of i and j in volume data, we prepared them in advance by an array (see the argument of *vecIJ*). Each work-group processes all voxels in a single vertical line and thus the index of i and j can be shared by all work-items in a work-group. We use local memory to achieve this goal as lines 5~8 shown.

(iii) In line 6, we use local memory to cache the shareable data of F, X, and F2, which are defined the same to the variables in line 6 of Listing 7. Note that the used batch number (namely B) ranges from 1~32 in our implementations. Hence, the required size of local memory is sizeof(float)*B*3, which is greatly small than the capacity of local memory. All of these values are only computed once as in lines 12~18 and reused by all work-items in a work-group. Note that a barrier is required as in line 18 to synchronize all work-items in a work-group.

(iv) Using the proposed sub-line algorithm (as in Section 3.2.1), we perform the linear interpolation via fast local memory as in lines 24~30. More importantly, we can vectorize the computation as Figure 2b shows.

(v) Relying on the pixel values in local memory (namely the variable of sMem), the second linear interpolation can be

Listing 8: The OpenCL kernel for back-projection. The constant memory is used to store the projection matrix (see mat), global memory is used to store the projections (see img) and volume data (see volume), the imgDim (namely width, height, and nProj) and volDim (namely nx, ny, and nz) respectively. ROWS is a constant value of 3. LM_SIZE equal imgDim.x (the width of projections). B is defined as the batch number and equals to imgDim.z.

```
1  __kernel void symmetry_lm_cl(__constant float4* mat,
2    __global float* img, int3 imgDim, __global float* volume,
       int3 volDim, __global int2* vecIJ)
3  {
4    int k = get_global_id(0); //global index x
5    __local int2 ij; //share index i and j
6    __local float F[B], X[B], F2[B], sMem[LM_SIZE];//local mem
7    if (k == 0)
8      ij = vecIJ[get_global_id(1)]; //i and j ← global index y
9    barrier(CLK_LOCAL_MEM_FENCE); //barrier for local memory
10
11   float4 ijkw = (float4)(ij.x, ij.y, k, 1.f); //vector (i,j,k)
12   if (k < nProj) {
13     float z = 1.f / dot((mat + ROWS * k)[2], ijkw);//compute z
14     X[k] = dot((mat + ROWS * k)[0], ijkw)*z; //compute x
15     F[k] = z; //cache z
16     F2[k] = z * z; //compute weight
17   }
18   barrier(CLK_LOCAL_MEM_FENCE); //barrier for local memory
19
20   int SIZE = imgDim.x * imgDim.y; //the size of a projection
21   float2 sum = (float2)(0.f, 0.f);
22   for (int s = 0; s < B; s++, img += SIZE, mat += ROWS){
23     //sub-line algorithm as in Fig. 2
24     int nx = convert_int(X[s]);
25     float dx = X[s] - convert_float(nx);
26     __global float* ptr0 = img + nx * imgDim.x; //see Fig. 2a
27     __global float* ptr1 = ptr0 + imgDim.x; //see Fig. 2a
28     for (int m = k; m < width; m += get_local_size(0))
29       sMem[m] = mix(ptr0[m], ptr1[m], dx); //see Fig. 2b
30     barrier(CLK_LOCAL_MEM_FENCE); //barrier for local memory
31
32     //y and _y are symmetric at the vertical line of FPD
33     float y = dot(mat[1], ijkw)*F[s];
34     float _y = width - 1 - y;
35     {
36       int2 ny = convert_int2((float2)(y, _y));//float → int
37       float2 dy = (float2)(y, _y) - convert_float2(ny);
38       //linear interpolation and update sum
39       sum += mix((float2)(sMem[ny.x], sMem[ny.y]), (float2)(
           sMem[ny.x+1], sMem[ny.y+1]), dy)*F2[s];
40     }
41     barrier(CLK_LOCAL_MEM_FENCE); //barrier for local memory
42   }
43   int offset = ij.y*volDim.z*volDim.x + ij.x * volDim.z;
44   volume[offset + k] += sum.x; //update volume
45   volume[offset + volDim.z - 1 - k] += _sum.y; //update volume
46 }
```

performed as in line 30.

(vi) This kernel also employs geometric symmetry to simply the computations as in lines 33~34 and lines 44~45 correspondings to codes in Listing 7.

We use two dimensional NDRange to launch this kernel, the local and global parameters may be written as (nz/2, 1) and (nz/2, sizeIJ), respectively. Note that sizeIJ is the number of elements in the array of vecIJ (as in arguments). We emphasize the intrinsic that get the id of work-items in bold font, e.g. get_global_id, get_local_size, etc. Several built-in intrinsic is used in our implementation such as dot, mix, convert_T. More explanations on these intrinsic can refer to [29];

3.2.1 Sub-line algorithm

This section illustrates our sub-line algorithm. In Figure 2, we show the details of vectorized operations on memory access and arithmetic computations. As shown in Figure 2a, the variables of ptr0 and ptr1 represent the addresses of two neighboring lines in a projection. The computations on obtaining these two addresses can be found in lines 26~27 of Listing 8. We load data from global memory in a coalesced pattern that meets the caching mechanism of global memory and thus, the wide SIDM intrinsic can be employed. We use the vload8 and vstor8 as an illustrative example in Figure 2b. Note that such OpenCL built-in intrinsic is corresponding to the SIMD intrinsic such as SSE, AVX [30]. we can use wider intrinsic (e.g. vload16, vload32, etc.) due to the improved data layout using transposed projections and volume data as introduced in Section 3.1.1. Such kind of vector instruction set can be be automatically generated via compilation option in Intel OpenCL compiler of *ioc* such as "-simd=avx" or "-simd=sse42".

The local memory (as the variable of *sMem* in line 29 of Listing 8 or Figure 2a) is used to cache a line of projection at subpixel precision. It is because we use the *mix* intrinsic to blend the two lines of projections (namely the ptr0 and ptr1) into a line of data. We benefit from this algorithm as follows: Firstly, all elements in these two lines only accessed once in a regular pattern; Secondly, one of the linear interpolations can be well parallelized (or blended). Thirdly, the second linear interpolation can be performed via cache-optimized local memory and thus avoid reading the slow global memory. Since we only cache a line of projection data, the number of elements is the width of the projection (as the variable of LM_SIZE in Listing 8). Note that We define this variable dynamically as a compilation option to the kernel.

3.2.2 Cache prefetching methodology

This section describes the cache prefetching methodology. We propose a novel algorithm to overlap the loading operation and computation by dual buffering technique. As Algorithm 1 shown, we declare dual local memory in line 1 (as emphasized in gray color). Hence, we can perform loading operation(as in line 6) and computation (as in line 8) using different buffers. This methodology benefits from the local memory. However, the twice sizes of local memory will be used and thus, the available size of processing projections will be limited by the capacity of local memory. The constrain may be written as $sizeof(float) * 2 * LM\_SIZE < 32K$. Hence, the LM_SIZE (or the width of the transposed projections) should be less than 4096.

4. Evaluation

In this section, we introduce the evaluation environment, report the evaluated performance, and discuss the advantages and limitations of proposed algorithms.

4.1 Experiment setup

As Table 1 shown, we use four kinds of Intel CPUs to evaluate our implementation including the Operating System (OS) and the capacity of the host memory. The OpenCL SDK-2019.5.345 and runtime 18.1.0 are used for developing and running OpenCL kernels. All OpenCL kernels are compiled with option as "-cl-mad-enable -cl-fast-relaxed-math". GCC 5.4 is used for compiling the

Algorithm 1: Cache prefetching by dual local memory. All comments about lines refer to Listing 8.

```
   Input  : .....          ▷ the input argument as same to Listing 8
   Output : volume         ▷ the output volume
1  __local sMem[2][LM_SIZE]   ▷ declear dual scratchpad memories
2  sMem[0] ← img[0]           ▷ prefetching by sub-line alg. as in line 29
3  sum ← 0                    ▷ declear registers as in line 21
4  for i = 0 to B − 1 do
5      if i + 1 < B then
6          sMem[(i+1)%2] ← img[i + 1]   ▷ prefetching without barrier
7      sum ← sMem[i%2]        ▷ compute via scratchpad as in line 39
8      barrier(CLK_LOCAL_MEM_FENCE)   ▷ barrier as in line 41
9  volume ← sum               ▷ update volume as in lines 44~45
```

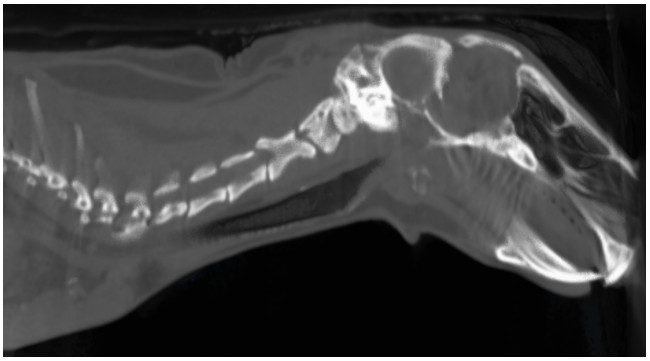


Fig. 3: The reconstructed volume data using open source data in RabbitCT [7].

Table 1: Evaluation environment.

Intel CPU	Xeon E5-2630 v4	Xeon E5-2650 v3	Xeon Gold 6140	Core i7-9700K
Cores	10	10	18	8
Threads/core	2	2	1	1
Frequency	2.2GHz	2.3GHz	2.3GHz	3.6GHz
Sockets	2	2	2	1
Memory	256GB	256GB	192GB	32GB
OS	CentOS 7.4	CentOS 7.4	CentOS 7.4	Ubuntu 16.04

host codes, "-O3 -fopenmp -lpthread -std=c++11 -march=native -fno-tree-vectorize -fno-tree-slp-vectorize" is adopted for compilation. Since the arithmetical computation is independent of the content of projections and volume data, we use volume data of RabbitCT [7] as in Figure 3 to generate a wide variant of projections by a forward-projection tool in RTK library [22]. The sizes of the experimental projections include 256^2, 512^2, 1024^2. The number of projections is fixed as 512.

In Table 2, we list a collection of back-projection kernels using algorithms as illustrated in Section 3. We take advantage of both OpenMP and OpenCL to optimize back-projection, the characteristics of employed optimization techniques are named as "Transpose", "Share", "Symmetry", "Scratchpad", "LocalMem", and "Prefetching". Regarding these techniques, the "Transpose" corresponds to the algorithm in Listing 4, "Share" in Listing 5, "Symmetry" in Listing 6, "Scratchpad" in Listing 7. "LocalMem" indicates the use of sub-line algorithm in Figure 2a in OpenCL-optimized implementation. "Prefetching" means the use of technique in Algorithm 1, which uses local memory via OpenCL.

Table 2: Back-projection Kernel names and employed optimizations.

API	Name	Transpose	Share	Symmetry	Scratchpad	LocalMem	Prefetching
OpenMP	baseline						
	transpose_mp	✓					
	share_mp	✓	✓				
	symmetry_mp	✓	✓	✓			
	scratchpad_mp	✓	✓	✓	✓		
OpenCL	transpose_cl	✓					
	share_cl	✓	✓				
	share_lm_cl	✓	✓			✓	
	symmetry_cl	✓	✓	✓			
	symmetry_lm_cl	✓	✓	✓		✓	
	symmetry_pf_cl	✓	✓	✓		✓	✓

4.2 Results & Discussion

In this section, we evaluate the performance of the kernels as listed in Table 2 and discuss the performance gains according to the characteristics of the implementations.

4.2.1 Impact of batched processing

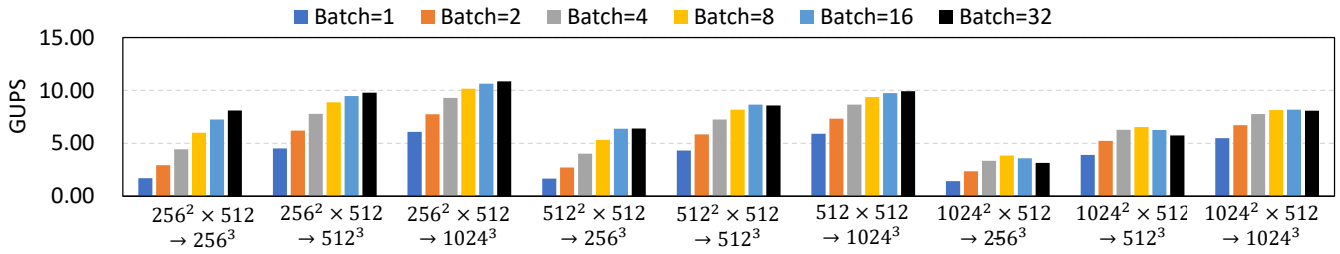
This section discusses the impact of batched processing on OpenCL-optimized back-projection kernels. We do not show the performance impacts on the OpenMP-optimized algorithms due to their performances are independent of such processing. Taking Listing 1 for an instance, the pragma syntax in line 4 is inside the loop of batched processing. Using our fastest kernel (namely symmetry_pf_cl in Table 3) as an illustrative example, we show its performance with different configurations of batch numbers in Figure 4. It is clear that the performance increases greatly with a smaller ratio of problems (as defined in Section 2.3), e.g. the performance of problem $256^2 \times 512 \rightarrow 256^3$ on all CPUs is nearly linear to the number of batches. It is because using a large batch help to improve the data locality and reduce global memory access.

4.2.2 Performance on CPUs

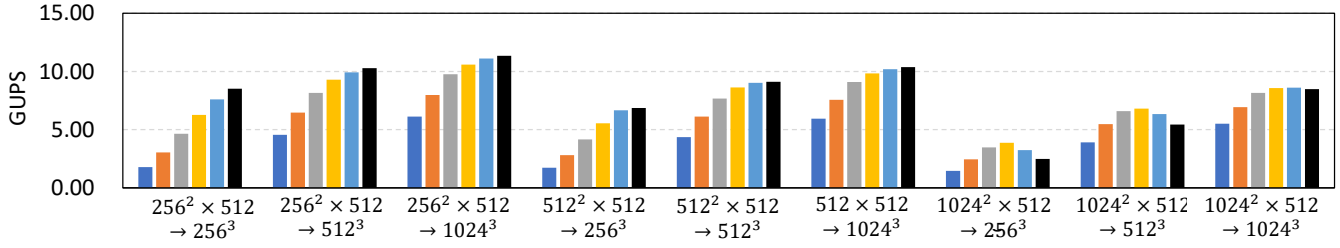
As shown in Table 3, we report the performance of several back-projection kernels on different CPUs. The evaluated problems are as follows: the sizes of input problems are 256^2, 512^2, and 1024^2; the sizes of output problems are 256^3, 512^3, and 1024^3. The baseline implementation can refer to Listing 1, the characteristics of all evaluated kernels can be found in Table 2. Among OpenMP-optimized algorithms, the symmetry_mp achieves the highest performance due to the best uses of optimizing techniques. Mostly, the symmetry_pf_cl achieves the best performance as emphasized in blue color. Comparing to the results of baseline implementations as emphasized in blue color, the symmetry_pf_cl performs up to 10× speed up. It is noteworthy that a part of the ratio of speed up can be found in Figure 5.

4.2.3 Discussing optimizations

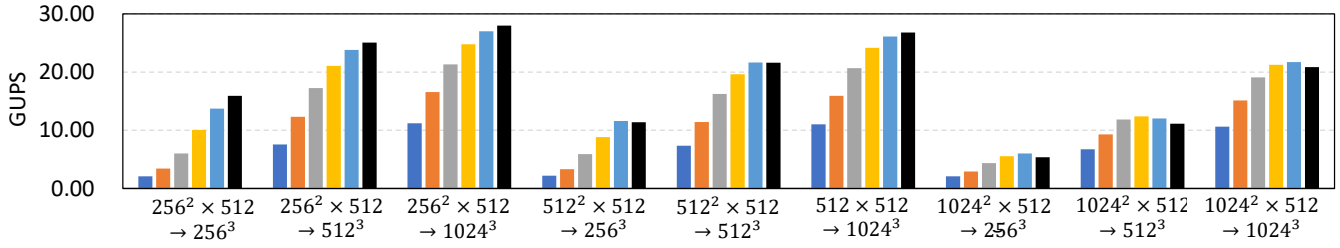
In Figure 5, we display the detailed speed up the performance of seven kernels by comparing to the kernel of baseline_mp. We can conclude as follows:



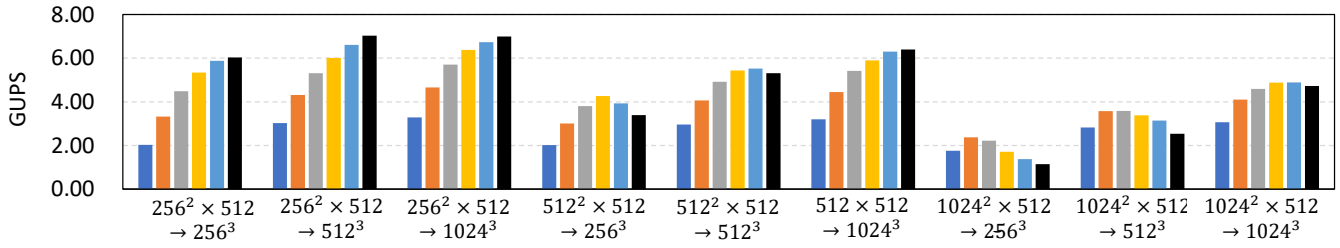
(a) Performance of symmetry_pf_cl on dual E-2630 CPUs.



(b) Performance of symmetry_pf_cl on dual E-2650 CPUs.



(c) Performance of symmetry_pf_cl on dual Gold 6140 CPUs.



(d) Performance of symmetry_pf_cl on a single i7-9700K CPU.

Fig. 4: Performance evaluation on kernel of symmetry_pf_cl with a variant of batch number, i.e. 1, 2, 4, . . . , 32.

(i) OpenCL-based kernels outperform the OpenMP-based kernels due to the speed up by OpenMP-optimized kernels is less than 1.5× and OpenCL-optimized kernels can up to 10×, e.g. symmetry_mp and symmetry_pf_cl.

(ii) The cache prefetching methodology (as in Section 3.2.2) is effective to improve the OpenCL kernels. We can find out the symmetry_pf_cl performs better than symmetry_pf_cl in a variant of CPUs and problems.

(iii) Sub-line algorithm (as in Section 3.2.1) is very effective to improve the performance of back-projection kernels. It is clear that both share_lm_cl and symmetry_lm_cl perform better than share_cl and symmetry_cl, respectively.

(iv) The used techniques such as sharing data and geometry symmetry as illustrated in Section 3.1 contribute to improving the performance of back-projection kernels. Obviously, share_cl, share_lm_cl, and symmetry_cl outperform the transposal_cl.

5. Related work

Back-projection is a rich-researched computing kernel. Target-specified hardware is often adopted to speed up the computation of back-projection. In [3], Wu et al. used Application-Specific Integrated Circuits (ASIC) to speed up the back-projection algorithm. The authors in [4], [5], [31], [32] also employed FPGA to tune the computation of FDK algorithm. Recently, there exists a trend as in [4], [5], [31], [32] to use high-level synthesis method, e.g. OpenCL, to generate the fast back-projection kernels on FPGA accelerator rather than using HDL language for lower the developing cost. In [18], [33], the authors employed the SIMD instruction set extensions to speed up the FDK computation and achieved outstanding performance on CPUs. In [34], Johannes et al. optimized the *RabbitCT* benchmark on the Intel Xeon Phi accelerator as performance engineering. Using a fixed-point DSP (Digital Signal Processor) platform, Liang et al. presented an optimized implementation of the FDK and achieved state-of-the-art

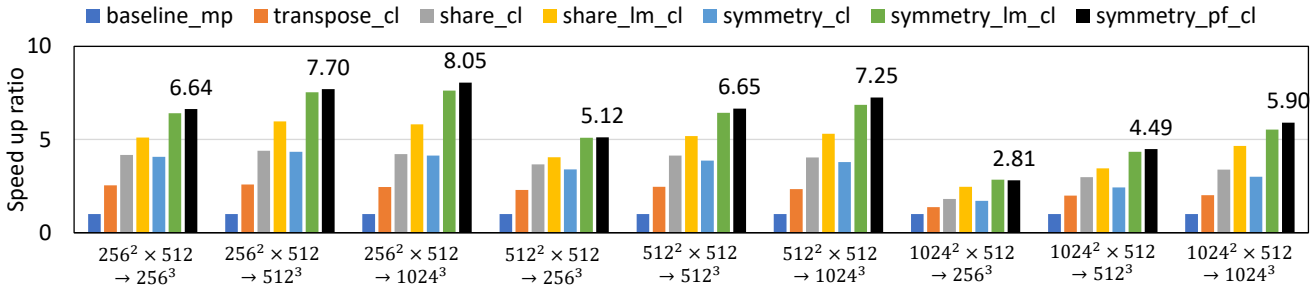
Table 3: Performance of back-projection on CPUs by OpenMP and OpenCL in an unit of GUPS. The batch number is fixed as 32.

	CPU	kernel	Performance (GUPS)								
			$256^2 \times 512 \rightarrow 256^3$	$256^2 \times 512 \rightarrow 512^3$	$256^2 \times 512 \rightarrow 1024^3$	$512^2 \times 512 \rightarrow 256^3$	$512^2 \times 512 \rightarrow 512^3$	$512^2 \times 512 \rightarrow 1024^3$	$1024^2 \times 512 \rightarrow 256^3$	$1024^2 \times 512 \rightarrow 512^3$	$1024^2 \times 512 \rightarrow 1024^3$
OpenMP	E5-2630×2	baseline	1.22	1.27	1.35	1.25	1.29	1.37	1.12	1.28	1.37
		transpose_mp	1.05	1.23	1.32	1.04	1.24	1.32	1.03	1.27	1.33
		share_mp	1.45	1.63	1.75	1.41	1.59	1.74	1.21	1.64	1.77
		symmetry_mp	1.52	1.67	1.84	1.50	1.71	1.84	1.48	1.68	1.86
		scratchpad_mp	1.30	1.43	1.54	1.36	1.50	1.53	0.89	1.10	1.25
	E5-2650×2	baseline	1.25	1.30	1.38	1.29	1.30	1.41	1.22	1.31	1.41
		transpose_mp	1.08	1.27	1.36	1.08	1.27	1.36	1.04	1.28	1.36
		share_mp	1.43	1.67	1.81	1.42	1.68	1.81	1.35	1.66	1.81
		symmetry_mp	1.54	1.73	1.91	1.55	1.78	1.91	1.53	1.75	1.93
		scratchpad_mp	1.46	1.56	1.61	1.43	1.56	1.61	0.92	1.12	1.30
	Gold-6140×2	baseline	2.52	2.65	2.63	2.52	2.64	2.78	2.31	2.58	2.72
		transpose_mp	2.43	2.59	2.64	2.38	2.53	2.69	2.28	2.53	2.65
		share_mp	3.18	3.44	3.49	3.14	3.46	3.58	2.91	3.40	3.50
		symmetry_mp	3.41	3.29	3.40	3.20	3.41	3.37	3.02	3.49	3.55
		scratchpad_mp	2.19	2.81	2.85	1.71	2.80	2.96	1.67	1.99	2.92
	i7-9700K×1	baseline	0.95	0.95	0.93	0.91	0.93	0.93	0.88	0.92	0.92
		transpose_mp	0.91	0.92	0.91	0.89	0.90	0.91	0.89	0.90	0.90
		share_mp	1.29	1.30	1.30	1.27	1.28	1.29	1.25	1.27	1.29
		symmetry_mp	1.31	1.20	1.18	1.31	1.22	1.19	1.29	1.28	1.24
		scratchpad_mp	0.91	0.93	0.92	0.88	0.93	0.96	0.65	0.80	0.86
OpenCL	E5-2630×2	transpose_cl	3.10	3.28	3.30	2.87	3.17	3.21	1.53	2.55	2.76
		share_cl	5.09	5.58	5.70	4.58	5.33	5.53	2.02	3.82	4.64
		share_lm_cl	6.23	7.58	7.84	5.06	6.69	7.27	2.75	4.42	6.37
		symmetry_cl	4.97	5.51	5.59	4.24	4.99	5.19	1.91	3.11	4.11
		symmetry_lm_cl	7.82	9.56	10.2	6.37	8.30	9.40	3.18	5.56	7.58
		symmetry_pf_cl	8.10	9.78	10.8	6.40	8.58	9.93	3.15	5.75	8.08
	E5-2650×2	transpose_cl	3.23	3.40	3.43	3.03	3.30	3.33	1.49	2.61	2.84
		share_cl	5.34	5.86	5.94	4.92	5.64	5.79	1.88	3.81	4.72
		share_lm_cl	6.53	7.85	8.18	5.34	7.02	7.66	2.26	4.40	6.64
		symmetry_cl	5.15	5.64	5.69	4.48	5.17	5.25	1.84	3.19	4.22
		symmetry_lm_cl	8.23	10.0	10.8	6.74	8.83	9.90	2.48	5.36	7.97
		symmetry_pf_cl	8.52	10.2	11.3	6.85	9.11	10.3	2.48	5.44	8.48
	Gold-6140×2	transpose_cl	5.45	5.94	6.03	4.70	5.92	6.06	2.65	4.05	5.93
		share_cl	9.02	11.3	11.8	7.25	11.1	11.8	3.95	6.38	11.2
		share_lm_cl	13.1	17.7	19.4	10.1	15.6	17.9	4.94	10.2	15.4
		symmetry_cl	9.50	11.9	12.6	7.18	11.3	12.4	3.76	5.90	10.4
		symmetry_lm_cl	16.4	24.8	28.8	11.6	21.4	26.6	5.52	11.2	20.4
		symmetry_pf_cl	15.9	25.0	27.9	11.3	21.6	26.8	5.37	11.1	20.8
	i7-9700K×1	transpose_cl	2.00	1.92	1.91	1.87	1.88	1.90	1.06	1.58	1.72
		share_cl	3.05	3.14	2.93	2.76	3.14	2.98	1.19	2.28	2.58
		share_lm_cl	4.70	5.37	5.49	2.88	4.62	4.97	1.04	2.41	4.25
		symmetry_cl	3.51	3.63	3.34	2.54	3.22	3.16	1.12	1.97	2.54
		symmetry_lm_cl	5.63	6.34	6.36	3.34	5.03	5.89	1.14	2.57	4.58
		symmetry_pf_cl	6.03	7.03	6.99	3.39	5.31	6.40	1.15	2.54	4.72

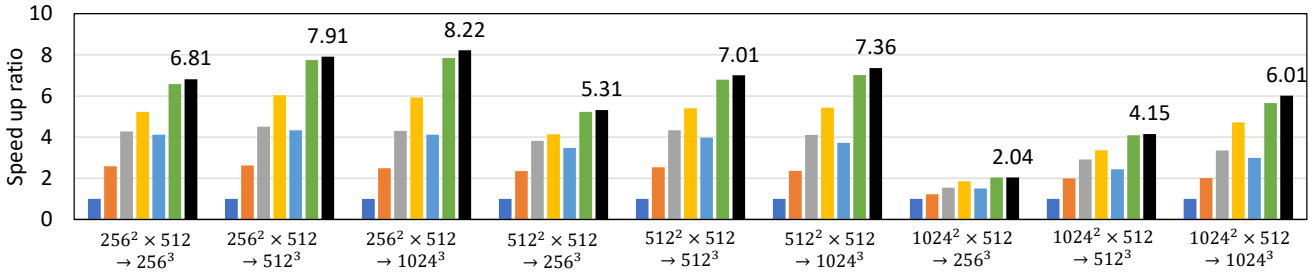
balance in cost and power consumption [6]. On several hardware accelerators platforms (i.e. CPUs, GPGPUs, Intel Xeon Phi), Serrano et al. [35] illustrated a parallelized FDK implementation. To solve the out-of-core problem, the authors in [10], [28] discussed decomposing the output problems into neighboured sub-volumes.

Using OpenCL is an effective approach to improve the performance of applications on multicore CPUs. Comparing to the naive vectorization, the authors in [36] demonstrated a novel dataflow conversion algorithm and achieved an average speedup factor of 2.5 on Intel CPUs. In [37], the authors demonstrated the performance advance of OpenCL on modern multicore CPUs.

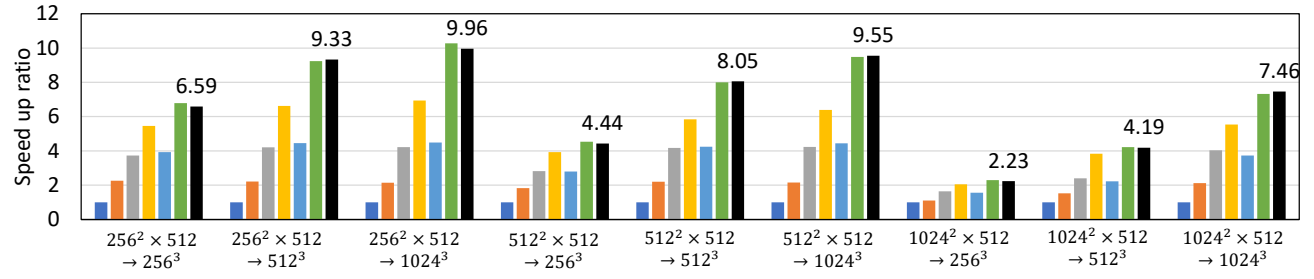
Regarding CT image reconstruction, the major requirements are lower radiation dosage, faster computation, and higher image quality. In practice, Filtered back-projection methodology is widely employed for image reconstruction due to the balanced and acceptable performance [1] that meets the major requirements. Unlike the prevalent approach that employs expensive accelerators (e.g. GPU, FPGA) to speed up the computation of the FDK algorithm, we directly use the commonly accessible Intel CPU (×86 architecture).



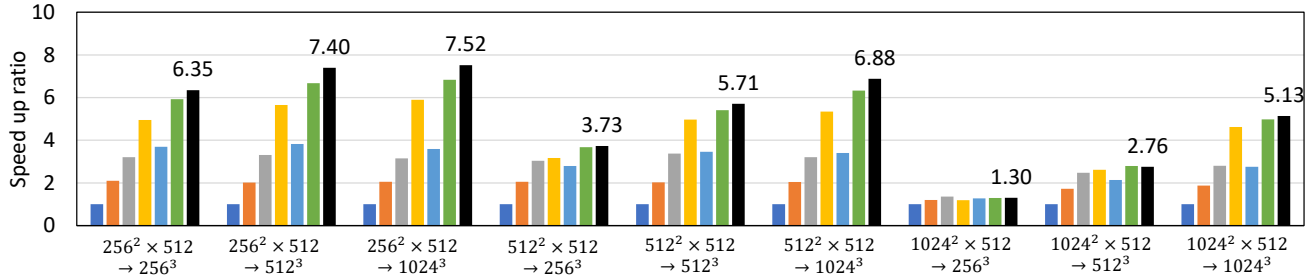
(a) Speed up evaluation on dual E-2630 CPUs.



(b) Speed up evaluation on dual E-2650 CPUs.



(c) Speed up evaluation on dual Gold 6140 CPUs.



(d) Speed up evaluation on a single i7-9700K CPU.

Fig. 5: Speed up evaluation on four kinds of Intel CPU. The baseline performance is from the function of baseline_mp.

6. Conclusion & future work

To leverage the use of OpenCL on CPUs, this paper demonstrates several algorithmic and paralleling optimizations for FDK algorithms on several Intel ×86 multicore processes. Our results show that the OpenCL-optimized back-projection performs up to 10 times faster than the multi-threading optimized implementation. Our implementations benefit from the high-level vectorization, best use of the constant/local memory in OpenCL, and regular memory access to the host memory. We innovate a sub-line algorithm to cache the projection for reducing the global memory access and reducing the arithmetic computations for bilinear interpolation. The proposed algorithms can be used in medical and industrial applications for high-resolution image reconstruction [38] due to its outstanding performance. Additionally, our methodology can be applied in heterogeneous accelerators, e.g. GPU, FPGA. In the future, we plan to implement our kernels using SYCL [39] for a wide variant of accelerators due to SYCL is a C++ abstraction layer on the top of OpenCL and can simplify the OpenCL programming. we also intend to apply OpenCL/SYCL in the medical image processing applications such as volume rendering [40].

Acknowledgment

This work was partially supported by JST-CREST under Grant Number JPMJCR19F5. Computational resource of AI Bridging Cloud Infrastructure (ABCI) provided by National Institute of Advanced Industrial Science and Technology (AIST) was used. We would like to thank Endo Lab at Tokyo Institute of Technology for providing a server. We would like to thank Dr. Jintao Meng at Tencent AI Lab for advising on experiments.

References

[1] Pan, X., Sidky, E. Y. and Vannier, M.: Why do commercial CT scanners still employ traditional, filtered back-projection for image reconstruction?, *Inverse problems*, Vol. 25, No. 12, p. 123009 (2009).

[2] Feldkamp, L., Davis, L. and Kress, J.: Practical cone-beam algorithm, *JOSA A*, Vol. 1, No. 6, pp. 612–619 (1984).

[3] Wu, M. A.: ASIC applications in computed tomography systems, *ASIC Conference and Exhibit, 1991. Proceedings., Fourth Annual IEEE International*, IEEE, pp. P1–3 (1991).

[4] Coric, S., Leeser, M., Miller, E. and Trepanier, M.: Parallel-beam backprojection: an FPGA implementation optimized for medical imaging, *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, ACM, pp. 217–226 (2002).

[5] Xue, X., Cheryauka, A. and Tubbs, D.: Acceleration of fluoro-CT reconstruction for a mobile C-Arm on GPU and FPGA hardware: a simulation study, *Medical Imaging 2006: Physics of Medical Imaging*, Vol. 6142, International Society for Optics and Photonics, p. 61424L (2006).

[6] Liang, W., Zhang, H. and Hu, G.: Optimized implementation of the FDK algorithm on one digital signal processor, *Tsinghua Science and Technology*, Vol. 15, No. 1, pp. 108–113 (2010).

[7] Rohkohl, C., Keck, B., Hofmann, H. and Hornegger, J.: RabbitCT—an open platform for benchmarking 3D cone-beam reconstruction algorithms, *Medical Physics*, Vol. 36, No. 9Part1, pp. 3940–3944 (2009).

[8] Xu, F. and Mueller, K.: Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware, *IEEE Transactions on nuclear science*, Vol. 52, No. 3, pp. 654–663 (2005).

[9] Rezvani, N., Aruliah, D., Jackson, K., Moseley, D. and Siewerdsen, J.: SU-FF-I-16: OSCaR: An open-source cone-beam CT reconstruction tool for imaging research, *Medical Physics*, Vol. 34, No. 6Part2, pp. 2341–2341 (2007).

[10] Zhao, X., Hu, J.-j. and Zhang, P.: GPU-based 3D cone-beam CT image reconstruction for large data volume, *Journal of Biomedical Imaging*, Vol. 2009, p. 8 (2009).

[11] Zinsser, T. and Keck, B.: Systematic performance optimization of cone-beam back-projection on the Kepler architecture, *Proceedings of the 12th Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, pp. 225–228 (2013).

[12] Wikipedia: X86 — Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=X86oldid=959376720 (2020). [Online; accessed 07-June-2020].

[13] Lee, J. H., Nigania, N., Kim, H., Patel, K. and Kim, H.: OpenCL performance evaluation on modern multicore CPUs, *Scientific Programming*, Vol. 2015 (2015).

[14] Nickolls, J. and Dally, W. J.: The GPU computing era, *IEEE micro*, Vol. 30, No. 2, pp. 56–69 (2010).

[15] Juille, H. and Pollack, J. B.: Parallel genetic programming and fine-grained SIMD architecture, *Working Notes for the AAAI Symposium on Genetic Programming*, pp. 31–37 (1995).

[16] Nickolls, J., Buck, I., Garland, M. and Skadron, K.: Scalable parallel programming with CUDA, *Queue*, Vol. 6, No. 2, pp. 40–53 (2008).

[17] Ali, A., Dastgeer, U. and Kessler, C.: OpenCL for programming shared memory multicore CPUs, *Fifth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2012) at HiPEAC-2012, 23 January, Paris, France*, HiPEAC Network of Excellence (2012).

[18] Treibig, J., Hager, G., Hofmann, H. G., Hornegger, J. and Wellein, G.: Pushing the limits for medical image reconstruction on recent standard multicore processors, *The International Journal of High Performance Computing Applications*, Vol. 27, No. 2, pp. 162–177 (2013).

[19] van Renen, A., Vogel, L., Leis, V., Neumann, T. and Kemper, A.: Persistent memory i/o primitives, *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pp. 1–7 (2019).

[20] Kak, A. C. and Slaney, M.: *Principles of computerized tomographic imaging*, IEEE press New York (1988).

[21] Hartley, R. and Zisserman, A.: *Multiple view geometry in computer vision*, Cambridge university press (2003).

[22] Rit, S., Oliva, M. V., Brousmiche, S., Labarbe, R., Sarrut, D. and Sharp, G. C.: The Reconstruction Toolkit (RTK), an open-source cone-beam CT reconstruction toolkit based on the Insight Toolkit (ITK), *Journal of Physics: Conference Series*, Vol. 489, No. 1, IOP Publishing, p. 012079 (2014).

[23] Wiesent, K., Barth, K., Navab, N., Durlak, P., Brunner, T., Schuetz, O. and Seissler, W.: Enhanced 3-D-reconstruction algorithm for C-arm systems suitable for interventional procedures, *IEEE transactions on medical imaging*, Vol. 19, No. 5, pp. 391–403 (2000).

[24] Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D. and McDonald, J.: *Parallel programming in OpenMP*, Morgan kaufmann (2001).

[25] Robison, A., Voss, M. and Kukanov, A.: Optimization via reflection on work stealing in TBB, *2008 IEEE International Symposium on Parallel and Distributed Processing*, IEEE, pp. 1–8 (2008).

[26] Sakamoto, C., Miyazaki, T., Kuwayama, M., Saisho, K. and Fukuda, A.: Design and implementation of a parallel pthread library (ppl) with parallelism and portability, *Systems and Computers in Japan*, Vol. 29, No. 2, pp. 28–35 (1998).

[27] Gaster, B., Howes, L., Kaeli, D. R., Mistry, P. and Schaa, D.: *Heterogeneous computing with openCL: revised openCL 1.*, Newnes (2012).

[28] Lu, Y., Ino, F. and Hagihara, K.: Cache-aware GPU optimization for out-of-core cone beam CT reconstruction of high-resolution volumes, *IEICE TRANSACTIONS on Information and Systems*, Vol. 99, No. 12, pp. 3060–3071 (2016).

[29] Munshi, A., Gaster, B., Mattson, T. G. and Ginsburg, D.: *OpenCL programming guide*, Pearson Education (2011).

[30] Lomont, C.: Introduction to intel advanced vector extensions, *Intel white paper*, Vol. 23 (2011).

[31] Subramanian, N.: A C-to-FPGA solution for accelerating tomographic reconstruction, PhD Thesis, University of Washington (2009).

[32] Henry, I. and Chen, M.: An FPGA Architecture for Real-Time 3-D Tomographic Reconstruction, PhD Thesis, University of California, Los Angeles (2012).

[33] Hofmann, J., Treibig, J., Hager, G. and Wellein, G.: Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi-and manycore chips, *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, ACM, pp. 57–64 (2014).

[34] Hofmann, J., Treibig, J., Hager, G. and Wellein, G.: Performance engineering for a medical imaging application on the Intel Xeon Phi accelerator, *Architecture of Computing Systems (ARCS), 2014 Workshop Proceedings*, VDE, pp. 1–8 (2014).

[35] Serrano, E., Bermejo, G., Blas, J. G. and Carretero, J.: High-performance X-ray tomography reconstruction algorithm based on heterogeneous accelerated computing systems, *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, IEEE, pp. 331–338 (2014).

[36] Karrenberg, R. and Hack, S.: Improving performance of OpenCL on CPUs, *International Conference on Compiler Construction*, Springer, pp. 1–20 (2012).

[37] Lee, J. H., Patel, K., Nigania, N., Kim, H. and Kim, H.: OpenCL performance evaluation on modern multi core CPUs, *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, IEEE, pp. 1177–1185 (2013).

[38] Chen, P., Wahib, M., Takizawa, S., Takano, R. and Matsuoka, S.: IFDK: A Scalable Framework for Instant High-Resolution Image Reconstruction, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3295500.3356163 (2019).

[39] Da Silva, H. C., Pisani, F. and Borin, E.: A comparative study of SYCL, OpenCL, and OpenMP, *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, IEEE, pp. 61–66 (2016).

[40] Drebin, R. A., Carpenter, L. and Hanrahan, P.: Volume rendering, *ACM Siggraph Computer Graphics*, Vol. 22, No. 4, pp. 65–74 (1988).