Fast Interpolation of Grid Data at a Non-Grid Point

Hiroshi Inoue
IBM Research - Tokyo
Tokyo, Japan
inouehrs@jp.ibm.com

***Abstract*—Defining data at a non-grid point by interpolating grid data is a common operation in many workloads including scientific applications and imaging applications. This paper describes our technique to accelerate this interpolation operation and show its performance benefit using 3D computed tomography reconstruction. The 3D CT is one of the compute-intensive medical imaging applications that frequently interpolates grid data (2D images) at a non-grid point. To efficiently execute this operation with SIMD instructions, we create an in-memory pre-computed table from the input 2D image at runtime before projecting voxels onto each image to 1) reduce the amount of computation and 2) avoid non-contiguous memory accesses that attenuate the benefits of SIMD instructions. We implemented and evaluated our pre-computation technique using a bilinear interpolation and a 3rd-degree Lagrange interpolation on POWER8 processors; it yields up to 75% and 57% performance improvements in the RabbitCT benchmark for the two interpolation algorithms respectively.**

Keywords - grid-based interpolation, 3D CT reconstruction, back projection, SIMD

I. INTRODUCTION

Interpolation of grid data to obtain data at a non-grid point is a common operation in many workloads including imaging applications and scientific stencil applications. The interpolation operation is often quite costly especially for large grid data (such as high-resolution images). One reason is the large number of arithmetic instructions to calculate the interpolated value. Another reason is its scattered memory access pattern; reading values from the surrounding pixels requires non-contiguous memory accesses, which reduce the efficiency of SIMD instructions even with the hardware gather/scatter support of the latest processors. In this paper, we aim to accelerate the interpolation of grid data using a new algorithm. To make the benefits and overhead of our new algorithm clear, we conduct throughout evaluations of our technique using a three-dimensional (3D) computed tomography (CT) workload [1] with two interpolation algorithms.

3D CT is one of the medical imaging workloads that frequently interpolate grid data (2D images). It reconstructs a structure in a 3D volume by backprojecting multiple projection images from different angles into the 3D volume using the grid-based interpolation. Figure 1 shows a schematic overview of the system of 3D cone-beam CT. An X-ray point source and a flat-panel X-ray detector move around the 3D object to capture 2D projection images from different angles. RabbitCT [2] is an open framework for benchmarking the backprojection algorithms in the FDK algorithm [3], a widely used non-iterative CT reconstruction algorithm. For each projection image, the FDK algorithm projects each volume element (voxel) in the 3D volume onto the projection image based on the transformation matrix (projection matrix) and then obtains the value at the projected point by performing a bilinear interpolation of the neighboring four pixels. The interpolation is critically important for image quality but causes significant overheads in the computation time [4]. Due to the advances in imaging hardware, the size and resolution of input images are increasing; hence the computation performance for the 3D reconstruction is also becoming more important to keep the overall CT system usability.

In this paper, we describe our new technique to accelerate the interpolation operation and hence the entire CT workload by significantly reducing the number of arithmetic instructions required and avoiding non-contiguous memory accesses. For each input projection image, we create a pre-computed table in memory at runtime, and the reconstruction loop accesses the pre-computed table instead of the projection image. For realistic resolutions, the benefit of using the pre-computed table in the reconstruction phase is much larger than the overhead of pre-computation. We implemented and evaluated this new technique on a system with 2-socket POWER8 processors; our technique improved the performance of the RabbitCT benchmark by up to 75%. We also evaluated it using a 3rd-degree Lagrange interpolation by enhancing RabbitCT and observed up to 57% speedups.

The basic idea of our pre-computation technique can be applied to a much wider range of applications and hardware compared to those we evaluate in this paper. Although we evaluated our technique using a non-iterative CT reconstruction algorithm, it can be applied to other imaging applications. For example, some iterative CT reconstruction and image-registration algorithms are good targets for our technique because they also interpolate data frequently. The key metric of the performance benefits is the ratio of the number of grid points (pixels) in data and the number of executed interpolation operations. Furthermore, certain non-imaging stencil applications are also potential targets. For instance, particle-in-cell simulations of two-phase flows [5] frequently interpolate the surrounding grid (mesh) points to obtain the flow parameters at the locations of the particles. Hence, they are interesting targets for our pre-computation technique. Our technique does not use features specific to the processor; hence, it can be implemented on other CPUs or GPUs.

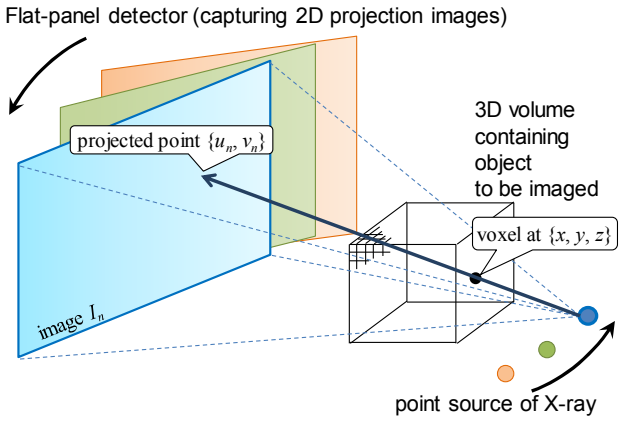

Figure 1. Overview of 3D cone-beam CT. Flat-panel detector and X-ray source move around 3D object to be imaged to capture 2D projection images from different angles.

II. Our Pre-Computation Technique

In this section, we first describe the baseline backprojection algorithm and details on how we enhance it to achieve higher performance by pre-computation.

A. Baseline Reconstruction Algorithm [2]

RabbitCT evaluates the performance and accuracy of the backprojection operation of the FDK algorithm [3] for 3D volume reconstruction. RabbitCT provides 496 projection images, whose size $S_x \times S_y$ is 1248×960 pixels, with a projection matrix for each image. The intensity value of each pixel is represented by a single-precision floating point value, and each element of the 3×4 projection matrix is represented by a double-precision floating point value. We denote the n-th projection image as I_n. For each of the projection images, the reconstruction algorithm projects all voxels onto I_n and updates the density value for the voxel based on the intensity value at the projected point obtained by interpolating the surrounding pixels. The number of voxels in the volume, i.e., the problem size, is $L^3 = 128^3$, 256^3, 512^3, or 1024^3. Among these four problem sizes, $L = 512$ is the most common, and $L = 128$ is a toy benchmark that is mostly used for debugging (hence it is not considered for ranking). Although different numbers of voxels are used, the same I_n are used as input for all problem sizes.

The voxel, whose 3D position is denoted as $\{x, y, z\}$, is projected onto point $\{u_n, v_n\}$ in I_n as follows. Here, a is a projection matrix determined by the system geometry and provided by the framework for each I_n.

$$\begin{aligned} u_n(x,y,z) &= (a_0x + a_3y + a_6z + a_9)/w_n(x,y,z), \\ v_n(x,y,z) &= (a_1x + a_4y + a_7z + a_{10})/w_n(x,y,z), \\ w_n(x,y,z) &= a_2x + a_5y + a_8z + a_{11}. \end{aligned} \tag{1}$$

I_n is defined only at grid points; hence, the intensity value at the projected point $\{u_n, v_n\}$, which we denote as $\hat{p}_n(u_n, v_n)$, is obtained by performing a bilinear interpolation from the intensity values p_n at the neighboring four pixels by

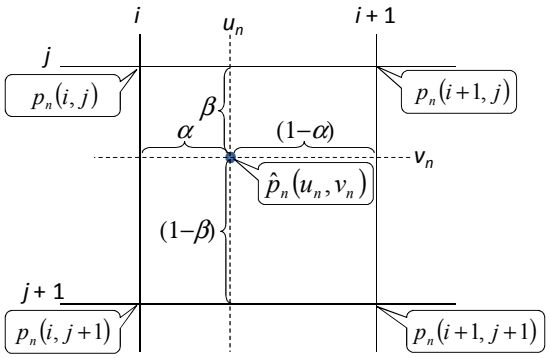

Figure 2. Overview of projected point $\{u_n, v_n\}$ and neighboring four grid points.

$$\begin{aligned} \hat{p}_n(u_n,v_n) &= (1-\alpha)(1-\beta)p_n(i,j) + \alpha(1-\beta)p_n(i+1,j) \\ &\quad + (1-\alpha)\beta p_n(i,j+1) + \alpha\beta p_n(i+1,j+1), \\ i &= \lfloor u_n \rfloor,\ j = \lfloor v_n \rfloor,\ \alpha = u_n - \lfloor u_n \rfloor,\ \beta = v_n - \lfloor v_n \rfloor. \end{aligned} \tag{2}$$

Here, $p_n(i, j)$ equals $I_n(i, j)$ if position $\{i, j\}$ is within I_n; otherwise, $p_n(i, j)$ is zero. Figure 2 shows an overview of the projected point and the neighboring pixels.

The density value of the voxel $f(x, y, z)$ is calculated as

$$f(x,y,z) = \sum_{n=1}^{N} \frac{1}{w_n(x,y,z)^2} \hat{p}_n(u_n,v_n) \tag{3}$$

Since this reconstruction algorithm has inherently huge parallelism, we can accelerate the execution of the algorithm by exploiting thread-level parallelism by multiple cores and data parallelism by SIMD instructions.

In this algorithm, computation performance in processor cores is the primary bottleneck. Since the accesses to the 2D images have spatial locality, i.e. neighbouring voxels are typically mapped onto the same or a nearby pixel in the 2D image, they do not cause too frequent cache misses. The accesses to the voxel data, which are mostly sequential, may become bottleneck depending on the memory system performance, but we can reduce this memory bandwidth to the voxel data by using a technique similar to temporal blocking [6, 7].

In addition to the execution time, RabbitCT reports on the errors from the reference implementation to evaluate the accuracy of the reconstruction algorithm. The reference implementation is a straightforward implementation of the algorithm using double-precision floating-point numbers during the computation.

B. Our Pre-Computation Technique

The bilinear interpolation with Equation 2 is the most time-consuming part of the overall execution time of the CT reconstruction. However, the interpolation is critically important for the overall quality of the reconstructed image; hence, we cannot simply skip the interpolation. In addition to the large number of arithmetic instructions to calculate Equation 2, vectorizing this equation is another high hurdle

with our pre-computation

```
1  for each projection image I_n (n = 0 to N - 1)
2  |  // pre-computation phase
3  |  for i = 0 to Sx - 1
4  |  |  for j = 0 to Sy - 1
5  |  |  |  calculate C_0 to C_3 by Equation 5 and store into pre-computed table
6  |  |  end
7  |  end
8  |  // reconstruction phase
9  |  for Iz = 0 to L - 1
10 |  |  for Iy = 0 to L - 1
11 |  |  |  determine range of Ix to iterate
12 |  |  |  for Ix = Ix_start to Ix_end
13 |  |  |  |  project voxel at (Ix, Iy, Iz) onto I_n
14 |  |  |  |  calculate p̂_n by Equation 4
15 |  |  |  |  update density of voxel
16 |  |  |  end
17 |  |  end
18 |  end
19 end
```

without our pre-computation

```
1  for each projection image I_n (n = 0 to N - 1)
2  |  // create zero-padded copy [4]
3  |  for i = 0 to Sx - 1
4  |  |  for j = 0 to Sy - 1
5  |  |  |  copy I_n(i, j) into separate memory buffer (Section 3.1)
6  |  |  end
7  |  end
8  |  // reconstruction phase
9  |  for Iz = 0 to L - 1
10 |  |  for Iy = 0 to L - 1
11 |  |  |  determine range of Ix to iterate
12 |  |  |  for Ix = Ix_start to Ix_end
13 |  |  |  |  project voxel at (Ix, Iy, Iz) onto I_n
14 |  |  |  |  calculate p̂_n by Equation 2
15 |  |  |  |  update density of voxel
16 |  |  |  end
17 |  |  end
18 |  end
19 end
```

R_L: resolution (size of a voxel), O_L: origin

Figure 3. Pseudocode of reconstruction with and without our pre-computation technique

for higher performance. It is known that vectorizing Equation 2 is inefficient with the SIMD instructions while it is almost straightforward to efficiently vectorize other parts of the algorithm including Equations 1 and 3 [4]. This inefficiency is due to the non-contiguous and unaligned memory accesses to read I_n from four pixels. A SIMD load instruction can load multiple elements only when they are contiguous in memory; hence, non-contiguous memory access incurs the overhead of executing multiple load instructions and mixing the loaded elements. Although some of the latest processors, such as Intel Haswell, support gather and scatter memory access instructions to load from or store to non-contiguous memory area, they are still slower than contiguous memory accesses.

To efficiently calculate Equation 2 by reducing the redundant computation and non-contiguous memory accesses, we group terms by u_n and v_n, the factors that depend on the current voxel location; we modify the equation as follows:

$$\begin{aligned}\hat{p}_n(u_n, v_n) &= (1-\alpha)(1-\beta)p_n(i,j) + \alpha(1-\beta)p_n(i+1,j) \\ &\quad + (1-\alpha)\beta p_n(i,j+1) + \alpha\beta p_n(i+1,j+1) \\ &= u_n v_n C_0(i,j) + u_n C_1(i,j) + v_n C_2(i,j) + C_3(i,j).\end{aligned} \tag{4}$$

Here, the coefficients C_0 to C_3 are calculated as

$$\begin{aligned}C_0(i,j) &= p_n(i,j) + p_n(i+1,j+1) \\ &\quad - p_n(i+1,j) - p_n(i,j+1), \\ C_1(i,j) &= (j+1)\big(p_n(i+1,j) - p_n(i,j)\big) \\ &\quad + j\big(p_n(i,j+1) - p_n(i+1,j+1)\big), \\ C_2(i,j) &= (i+1)\big(p_n(i,j+1) - p_n(i,j)\big) \\ &\quad + i\big(p_n(i+1,j) - p_n(i+1,j+1)\big), \\ C_3(i,j) &= (i+1)(j+1)p_n(i,j) + ij\,p_n(i+1,j+1) \\ &\quad - (i+1)j\,p_n(i,j+1) - i(j+1)p_n(i+1,j).\end{aligned} \tag{5}$$

The key point is that these coefficients C_0 to C_3 for each pixel in the 2D projection image are independent of u_n and v_n (hence independent of x, y, and z) by the definition. Hence, we can pre-compute these coefficients before executing the reconstruction and store the results in an in-memory pre-computed table. Since the number of voxels (L^3) is much larger than the number of pixels in a projection image ($S_x \times S_y$), multiple voxels are projected onto the same pixel position $\{i, j\}$. Hence, we have a huge opportunity to reuse the computed coefficients for interpolation. Once we pre-compute the coefficients for each pixel of the 2D input image, we can use lightweight interpolation with Equation 4. Equation 4 can be computed much more efficiently than Equation 2; it can be computed using only three multiply-and-add instructions as

$$\hat{p}_n(u_n, v_n) = u_n\big(v_n C_0(i,j) + C_1(i,j)\big) + \big(v_n C_2(i,j) + C_3(i,j)\big),$$

while Equation 2 requires 12 instructions to be computed in our implementation even with the multiply-and-add instructions. Figure 3 shows the pseudocode of the entire reconstruction algorithm with and without our pre-computation.

Our pre-computation technique uses additional memory space for the pre-computed table. The size of the table is four times as large as I_n because we hold four values, $C_0(i, j)$ to $C_3(i, j)$, per pixel for the bilinear interpolation. However, the overheads in memory space and in memory bandwidth are not significant compared to the space and bandwidth for other data structures, especially the density values of voxels. Access to the pre-computed table causes more cache misses due to its larger footprint than that of I_n. However, the benefit of reduced computations was much more significant than the drawback of an increased number of cache misses.

In addition to the reduced computation, which is beneficial regardless of the processor type, another benefit of using the pre-computed table is SIMD friendliness. By storing C_0 to C_3 for each pixel contiguously in memory aligned on 128-bit

address boundaries, we can totally avoid non-contiguous or unaligned memory accesses for I_n because these four values are enough to compute $\hat{p}_n(u_n, v_n)$, as shown in Equation 4, and we do not need to access I_n after the pre-computation phase. To fully exploit the data parallelism of SIMD instructions, each iteration of the inner-most loop of our algorithm loads C_0 to C_3 for four points by using four vector load instructions and then transposes the values in vector registers using the permutation instructions to pack the same coefficient of four points in one vector register. This transposition is also used by a previous implementation [4] and our baseline implementation for efficient use of SIMD instructions.

Although C_0 to C_3 stored in the pre-computed table are single-precision numbers, we execute the pre-computation phase using double-precision numbers and convert the resulting numbers to single precision numbers just before storing them in the pre-computed table. When we use single-precision numbers during the pre-computation, we observe degradation in accuracy. This is because rounding errors in the coefficients are magnified by multiplying by u_n and v_n in Equation 4. This degradation is not significant when we use double-precision numbers in the pre-computation phase.

C. Application to Higher Degree Interpolation Algorithms

We can apply the basic idea of our pre-computation to higher degree interpolation algorithms. In many scientific and imaging applications, bilinear interpolation cannot provide sufficient accuracy; hence, higher order interpolation algorithms are often used in real-world applications [5]. In this paper, we implemented and evaluated a 3rd-degree Lagrange polynomial interpolation, which uses 4×4 pixels (instead of 2×2 pixels of the bilinear interpolation) to obtain the interpolated value. For the 3rd-degree Lagrange interpolation, we need 16 coefficients per pixel in the pre-computed table. In general, the number of coefficients is equivalent to the number of pixels used in the interpolation. Hence, for a Kth-degree polynomial interpolation of a D-dimensional grid, the number of coefficients per pixel is $(K+1)^D$.

When we naively apply the transformation of Equation 4 to higher order algorithms, we experience accuracy problems. With the 3rd-degree Lagrange interpolation, the interpolated value is calculated as $\hat{p}_n(u_n, v_n) = u_n^3 v_n^3 C_0(i, j) + u_n^3 v_n^2 C_1(i, j) + \ldots$, but products of u_n and v_n, such as $u_n^3 v_n^3$, can potentially become huge; hence, this formula suffers from huge floating-point errors. To avoid this problem, we modify the equation of the interpolation by grouping terms by α and β instead of u_n and v_n as follows:

$$\begin{aligned}\hat{p}_n(u_n, v_n) = {} & \alpha^3\beta^3 C_0(i,j) + \alpha^3\beta^2 C_1(i,j) + \alpha^3\beta C_2(i,j) + \alpha^3 C_3(i,j) \\ & + \alpha^2\beta^3 C_4(i,j) + \alpha^2\beta^2 C_5(i,j) + \alpha^2\beta C_6(i,j) + \alpha^2 C_7(i,j) \\ & + \alpha\beta^3 C_8(i,j) + \alpha\beta^2 C_9(i,j) + \alpha\beta C_{10}(i,j) + \alpha C_{11}(i,j) \\ & + \beta^3 C_{12}(i,j) + \beta^2 C_{13}(i,j) + \beta C_{14}(i,j) + C_{15}(i,j).\end{aligned} \tag{6}$$

Here, $\alpha = u_n - \lfloor u_n \rfloor$, $\beta = v_n - \lfloor v_n \rfloor$.

The coefficients C_0 to C_{15} can be obtained by grouping terms in the equation of the Lagrange interpolation by the numbers of α and β included in each term. For example,

$$\begin{aligned}C_0(i,j) = {} & \\ & -(-p_n(i-1,j-1)/6 + p_n(i,j-1)/2 - p_n(i+1,j-1)/2 + p_n(i+2,j-1)/6)/6 \\ & +(-p_n(i-1,j)/6 + p_n(i,j)/2 - p_n(i+1,j)/2 + p_n(i+2,j)/6)/2 \\ & -(-p_n(i-1,j+1)/6 + p_n(i,j+1)/2 - p_n(i+1,j+1)/2 + p_n(i+2,j+1)/6)/2 \\ & +(-p_n(i-1,j+2)/6 + p_n(i,j+2)/2 - p_n(i+1,j+2)/2 + p_n(i+2,j+2)/6)/6\end{aligned}$$

Since α and β are in the range between 0.0 and 1.0 by definition regardless of u_n and v_n, this equation does not suffer from huge floating-point errors. However, there is the drawback that this equation requires two additional vector arithmetic instructions to calculate α and β in the inner-most reconstruction loop, but this additional cost of the two instructions does not matter for higher order interpolation algorithms severely because they use a much larger number of instructions than simple bilinear interpolation. When we apply this accurate version of the pre-computation to the bilinear interpolation, we observed about 10% performance reductions in a trade-off for better accuracy.

In summary, we introduced two different approaches to improve the accuracy for the bilinear and the 3rd-degree Lagrange interpolation algorithms by reducing the floating-point errors. For the bilinear interpolation, we use double-precision floating point numbers in the pre-computation phase. It increases the cost of pre-computation by up to twice, but it incurs no overhead in the reconstruction phase. Since the cost of the pre-computation is quite small compared to the reconstruction, the additional cost of using double precision in the pre-computation is less than 1% of the total execution time for large problem sizes. For the 3rd-degree Lagrange interpolation, we use an alternative equation (6) by grouping terms by α and β instead of u_n and v_n. This approach yields better accuracy in trade for the higher overhead in the reconstruction phase compared to the first approach of using double-precision numbers only in the pre-computation phase; the alternative equation requires two additional instructions in the reconstruction loop to calculate α and β. However, the additional overhead is about 3% and is much less significant compared to the benefits of the pre-computation.

D. Performance Modeling

As we empirically show later, the vector unit utilization is the primary bottleneck in this workload; hence, the number of executed vector instructions is the key to model the execution performance. We first show the numbers of vector instructions included in the inner-most loops of the reconstruction phase with and without pre-computation to estimate the benefit of our pre-computation technique. Then, we discuss the number of vector instructions for the pre-computation phase to evaluate the overhead of our pre-computation technique.

Each iteration of the inner-most loop of the reconstruction phase (lines 13-15 in Figure 3) executes the following steps for four voxels at once using SIMD instructions:

i) project each voxel onto a 2D image (calculate u_n, v_n, w_n),
ii) calculate i, j to find indices in the pre-computed table,
iii) load data from the pre-computed table (or image I_n),
iv) transpose data for four voxels in vector registers,
v) execute interpolation, and
vi) update density values of the voxels.

Table 1. Number of instructions that consume vector unit resource in each step of inner-most loop

Step	bilinear interpolation		3rd-degree Lagrange interpolation	
	with pre-computation	without pre-computation	with pre-computation	without pre-computation
i) calculate u, v, w (equation 1)	6	6	6	6
ii) calculate i, j	5	6	6	6
iii) load from pre-computed table or 2D image	0 (4 aligned load instructions)	8 (8 unaligned load insts)	0 (16 aligned load insts)	16 (16 unaligned load insts)
iv) transpose in vector registers	8	8	32	32
v) execute interpolation	3	12	15	46
vi) update density values	3	2	2	2
total	**25**	**42 (34 + 8†)**	**61**	**108 (92 + 16†)**

Note that some additional vector instructions can be inserted by compiler for register copying etc. / † means for unaligned load instructions.

These steps are not unique to our implementation and quite similar to the previous implementations [4]. Table 1 summarizes the number of instructions that consume the vector unit resource in each step of our implementation for two interpolation algorithms. From Table 1, we estimate that our pre-computation technique gives 1.68x (=42/25) performance gain for the bilinear interpolation and 1.77x (=108/61) gain for the 3rd-degree Lagrange interpolation with large problem sizes. On the little-endian PowerPC platform, an unaligned load instruction consumes vector unit resource while an aligned load instruction does not [8]. If an unaligned load instruction does not consume the vector unit resource, the benefit of our pre-computation will be smaller. However, even in such a case, our pre-computation still provides significant reduction in vector unit utilization; we estimate the performance gain will be 1.36x (=34/25) for the bilinear interpolation and 1.51x (=92/61) for the 3rd-degree Lagrange interpolation. As we discussed later, these estimates explain our experimental results well especially for bilinear interpolation. Table 1 assumes a 128-bit SIMD instruction set such as VSX of POWER or SSE of x86. When the vector length becomes longer, such as AVX of x86, the relative cost of transposition is increased. For the bilinear interpolation with 256-bit SIMD instructions, for example, we estimate the transposition uses 12 instructions while the numbers of instructions for other steps are unchanged (assuming the processor provides sufficiently flexible permutation instructions). This increased cost of transposition may attenuate the benefit of our technique slightly, but our technique can still give good reduction in the vector instructions in the inner-most loop.

In the current implementation of the pre-computation phase for the bilinear interpolation, we use both single-precision and double-precision floating-point vector arithmetic instructions to keep the accuracy as already discussed. In total, we use 85 vector instructions (including two unaligned load instructions) for each iteration of the inner-most loop, which processes four pixels. For the 3rd-degree Lagrange interpolation, we use 124 single-precision arithmetic instructions (including four unaligned load instructions) per four pixels in the pre-computation loop.

From these numbers and the sizes of the input image and 3D volume, we estimate the overhead of pre-computation as follows. For the problem sizes of L = 512 and 1024, the ratios of the numbers of vector instructions consumed for the pre-computation are less than 3% and 0.5%, respectively, with either interpolation algorithm. Hence, the overhead caused by the pre-computation is much smaller than the benefit of the reduced computation time in the reconstruction phase unless the number of voxels is unrealistically small compared to the size of the projection images. As we experimentally show later, the ratios of the execution time of the pre-computation phase reasonably match these estimates.

We expect that the overhead of the pre-computation will become more insignificant on future CT systems that use higher resolution in both 2D projection images and 3D volume because the execution time of the pre-computation phase is proportional to the square of the 2D image resolution while the reconstruction phase follows the cube of the 3D volume resolution.

III. Implementation And Evaluations

In this section, we detail the implementation of our pre-computation technique for RabbitCT. Then, we show the experimental results on POWER8 processors to illustrate the benefit of our pre-computation technique. We implemented our technique on POWER8, but it does not use POWER-specific features; hence, it is applicable to other processors, such as Intel Xeon, or to GPUs.

A. Implementaion

We implemented the reconstruction algorithms in C++ and vectorized them by hand using the SIMD intrinsics provided by the IBM XL C++ compiler 13.1. We used single-precision floating point numbers in the implementation unless we explicitly denoted another data type. Because the vector registers of the underlying processor are 128-bit in length, one SIMD instruction can execute four operations for different values at once.

Our baseline implementation follows FastRabbit [4, 7], a state-of-the-art implementation for Intel processors. The baseline optimizations include: 1) replacing divide instructions with a reciprocal estimate instruction, 2) skipping the voxels that cannot be projected onto I_n in the inner-most loop, and 3) eliminating the conditional branches to check the out-of-image-bound accesses by creating a copy of I_n with

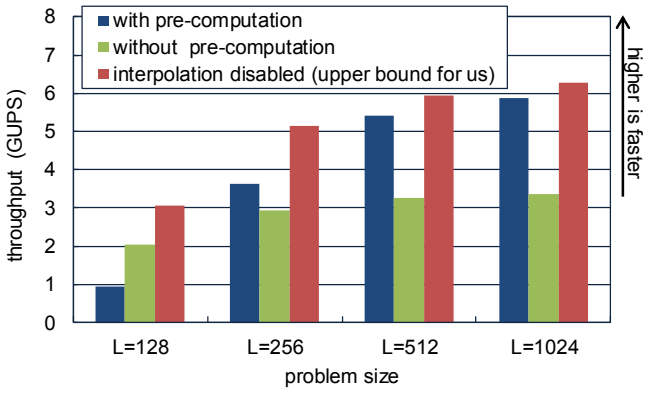

Figure 4. Performance (throughput in GUPS) with and without our pre-computation technique for bilinear interpolation on 5 cores (1 NUMA node) of POWER8.

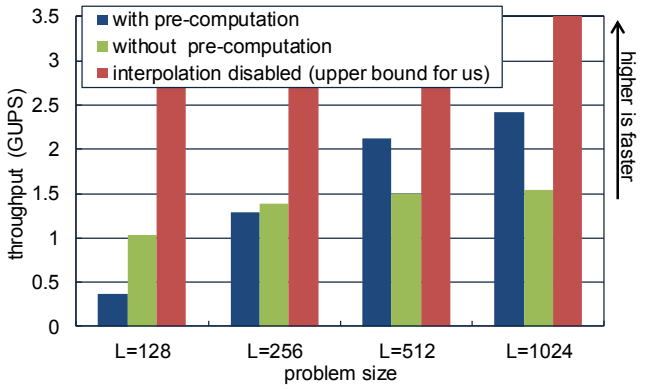

Figure 5. Performance (throughput in GUPS) with and without our pre-computation technique for Lagrange interpolation on 5 cores (1 NUMA node) of POWER8.

Table 2. Accuracy (root-mean-squared error in HU) with and without our pre-computation technique

Root-Mean-Square Error (HU)

Problem size	With pre-computation	Without pre-computation	Interpolation disabled
L=128	0.534	0.513	12.088
L=256	0.538	0.517	12.108
L=512	0.538	0.518	12.118
L=1024	0.545	0.526	12.120

zero padding around the image. These three optimizations were enabled even when we did not enable our pre-computation technique. In optimization 2, we solve the inequalities to determine the necessary range of I_x before executing the inner-most loop (line 11 in Figure 3). In optimization 3, we prepare a memory buffer of 1648×1000 pixels. I_n is copied into this buffer before executing the reconstruction (line 3-7 of "without pre-computation" in Figure 3). We use the same idea of padding zeros outside the image boundary to avoid conditional branches in the pre-computed table.

Many recent multi-socket systems have non-uniform memory architecture (NUMA); accesses to the directly attached (local) memory are faster than accesses to the remote memory attached to other sockets. Hence, it is important to reduce the remote memory accesses to achieve good performance scalability. To reduce remote memory accesses, we allocate the pre-computed table and density values of voxels for each NUMA node, and each worker thread is bound to a NUMA node. Then, all the threads in each NUMA node process different projection images rather than multiple NUMA nodes cooperating on one image. Hence, during the reconstruction phase, we do not need to access data in the remote memory. In the pre-computation phase, we might need to read an input 2D image from remote memory since we do not control the location of the input images. However, the size of an input image is much smaller than the density values of the voxels; hence, the accesses to the input images do not severely degrade the performance scalability. For better load balancing, each node picks a new input image from the global work queue after processing one input image rather than statically assigning images for each NUMA node. After processing all the projection images, we calculate the final density value for each voxel by summing up the partial results from all NUMA nodes. This final calculation is included in the execution time. Within a NUMA node, all threads cooperate to process one projection image; each thread picks a small block of voxels one by one and updates these values. An atomic operation (atomic add) is used only when a thread picks the next block from the node-local work queue. Because only one projection image is processed per node at a time, we do not need to use atomic operation to update values for voxels.

B. Evaluations

We evaluated our technique on POWER8 processors. The system has two 3.69-GHz POWER8 processors with 256 GB of system memory running Ubuntu Linux 14.10 for Little Endian POWER as its OS. The system has 20 processor cores (10 cores per socket), and the total number of SMT threads (logical CPUs) is 160 using an 8-way SMT. Since one POWER8 processor is a dual chip module, one socket is divided into two NUMA nodes; hence, each NUMA node is equipped with 5 cores (or 40 SMT threads). While we use all 8 SMT threads per core for experiments with the bilinear interpolation, we use only 4 SMT threads per core for the 3rd-degree Lagrange interpolation. This was because the SMT4 configuration outperformed SMT8 for the 3rd-degree Lagrange interpolation regardless of the use of the pre-computation. Each core is equipped with 256 KB L2 cache and 8 MB L3 cache memory. Also, the system has 256 MB shared L4 cache memory in total. We disabled the dynamic frequency scaling to obtain more consistent and reproducible results. We measured the performance 16 times and explain the average results.

Figure 4 and Table 2 illustrate the performance (throughput in GUPS, giga voxels updates per second) and accuracy (root-mean-square error from the reference result in

Table 3. Execution time breakdown on 5 cores of POWER8

Problem size	Linear interpolation				3rd-degree Lagrange interpolation			
	With pre-computation			Without pre-computation	With pre-computation			Without pre-computation
	pre-computation	recon struction	total		pre-computation	recon struction	total	
L=128	0.93 (47%)	1.11	2.01	0.94	2.81 (53%)	2.48	5.29	1.83
L=256	0.94 (22%)	3.36	4.19	5.21	2.83 (23%)	9.24	12.07	11.22
L=512	0.94 (4.1%)	22.20	22.59	37.90	2.84 (4.9%)	55.55	58.39	83.56
L=1024	0.93 (0.6%)	169.30	167.02	293.87	2.81 (0.7%)	410.13	412.94	649.50

- Numbers show execution time per projection image (in msec).
- Percentages shown in parenthesis show ratios of pre-computation to total execution time.

HUs, *Hounsfield units*) for bilinear interpolation with the problem sizes of L = 128, 256, 512, and 1024 with and without our pre-computation technique using one NUMA node equipped with 5 cores. GUPS is calculated by the number of voxels in the 3D volume (L^3) × the number of projection images (N) / execution time / 1024^3. It also illustrates the performance when we totally disabled the interpolation, i.e., we used $\hat{p}_n(u_n, v_n) = p(i, j)$ instead of Equation 2. This gives the upper-limit performance for our pre-computation technique, which reduces the overhead of the interpolation. Our pre-computation technique exhibited up to 75% performance improvements (for L = 1024) over the baseline (without pre-computation) without significant reduction in accuracy. When the interpolation was disabled, unlike with our technique, the accuracy was significantly degraded as a trade-off for higher performance. As described in Section II.B, we used double-precision numbers in the pre-computation phase. If we had used single-precision numbers, the accuracy would become as poor as about 2.75 HU. On the platform we used for evaluation, an unaligned load instruction consumes vector unit resource. To estimate the benefit of our pre-computation technique on platforms without such additional penalty in unaligned load instructions, we evaluated the performance without pre-computation by replacing all unaligned load instructions in the inner-most loop with aligned load instructions. This resulted in inaccurate results, but it allowed us to estimate the overhead due to the penalty in unaligned load instructions. Compared to this version, our pre-computation still gave up to 43% performance gains. These performance gains match the estimation from the performance model explained in Section II.D.

Figure 5 evaluates our pre-computation technique using the 3rd-order Lagrange interpolation. The gains due to our pre-computation were up to 57%. The gains were slightly lower than the estimation from the performance model. One possible reason for the difference between the measurements and the estimation is the overhead due to additional vector instructions for register copies inserted by the compiler since the higher degree interpolation algorithm involves more values and incurs higher register pressure. For the 3rd-order Lagrange interpolation, the changes in the accuracy caused by our pre-computation were negligible since we used an accurate version of the pre-computation with Equation 6.

The performance improvements with the pre-computation are more significant for a large problem size. When the problem size is tiny compared to the input size, e.g., L = 128 in Figures 4 and 5, the pre-computation degrades performance. Table 3 shows the execution time broken down into the pre-computation phase and reconstruction phase for each problem size. Because the size of the projection images is the same for all problem sizes, the time for the pre-computation phase is almost constant regardless of the problem size, while the larger problem sizes increase the time for the reconstruction phase. Therefore, the execution time of pre-computation is negligible for large problem sizes, but it matters for the total performance when the problem size is very small. These performance overheads almost match the estimations from the performance model. The estimations from the model were slightly smaller than the measured overhead partially because of a performance optimization in the reconstruction phase. We skip the voxels that cannot be projected onto the projection image in the inner-most loop of the reconstruction phase, but we do not consider this (data-dependent) optimization in the performance model.

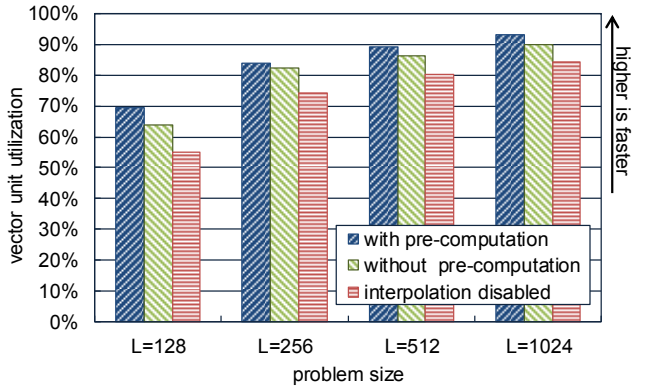

Figure 6. The vector unit utilization with and without our pre-computation technique for the bilinear interpolation on 5 cores of POWER8.

For more insight into the improvements with our pre-computation technique, we show the vector unit utilization and the number of cache misses measured by the performance counter of the processor. We select these two metrics because our technique reduces the number of vector instructions in a trade-off for increased cache misses.

Figure 6 shows the vector unit utilization for the bilinear interpolation. The utilization becomes 100% when one core

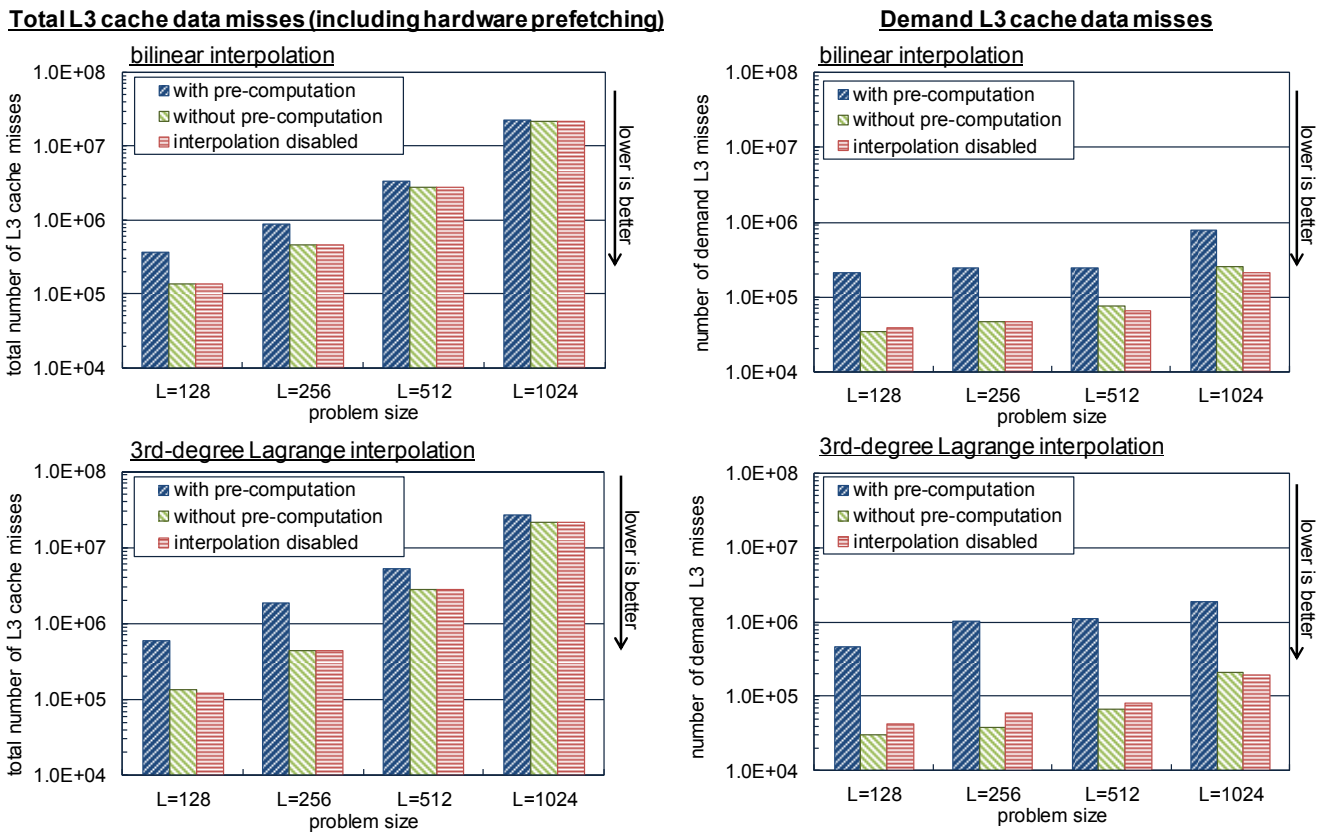

Figure 7. Total L3 cache misses (including cache misses caused by hardware prefetcher) and demand misses on 5 cores for bilinear interpolation and 3rd-degree Lagrange interpolation per input image. Y-axis is in logarithmic scale.

executes two vector instructions every cycle. As shown in the figure, the vector unit utilization is quite high: more than 85% for L = 512 and 1024 with or without pre-computation. This means that the vector unit is the primary bottleneck in this workload even with the increased cache misses with the pre-computation; hence, the performance gain due to the reduced vector instructions is much more significant than the overhead due to the increased cache misses.

Figure 7 compares the numbers of total L3 cache data misses and L3 cache demand data misses. We show the total misses, which includes cache misses caused by the hardware memory prefetcher, to evaluate how our technique affects the total amount of data fetched into the processors. We estimated the total cache misses caused by using the demand cache misses when the hardware memory prefetcher was disabled; hence, these estimates may slightly underestimate the real values. Our technique increased the L3 demand cache misses regardless of the problem size as a trade-off for reducing the number of executed vector instructions. The pre-computed table is 4x or 16x as large as the original 2D image; hence, the larger memory footprint of the pre-computed table increased demand cache misses. However, the increases in the total amount of data transfer were much less significant. These characteristics were common between two interpolation algorithms. The current algorithm reads and writes the density values of the all voxels for each projection image, and these accesses for the density values of voxels dominate the memory bandwidth of the system memory. Because the accesses to the voxel data are mostly sequential, the hardware memory prefetcher of the processor works well; the accesses to the voxel data do not cause frequent demand cache misses. Hence, the effect of our pre-computation on the amount of the data transfer between processors and memory is much less significant compared to the effects on the demand cache misses. If the memory bandwidth becomes the major performance bottleneck, we can use a technique similar to temporal blocking to reduce memory bandwidth requirements for accessing the voxel data [6, 7].

The POWER8 processor we used in the experiments supports 8 SMT threads per core, while other general-purpose processors typically support a smaller number of SMT threads; e.g. the latest x86 processors of Intel or AMD support 2-way SMT by hyper-threading. By increasing SMT level (threads per core), it is possible to hide the cache miss latency and improve the vector unit utilization. To confirm that our pre-computation is effective on other processors that only support a smaller number of SMT threads per core, we show the performance and the vector unit utilization for the bilinear interpolation using configurations with 1 to 8 SMT threads per core on POWER8 in Figure 8 and 9. Our pre-computation improved the performance regardless of the SMT level. The performance gain by the pre-computation was 57% with

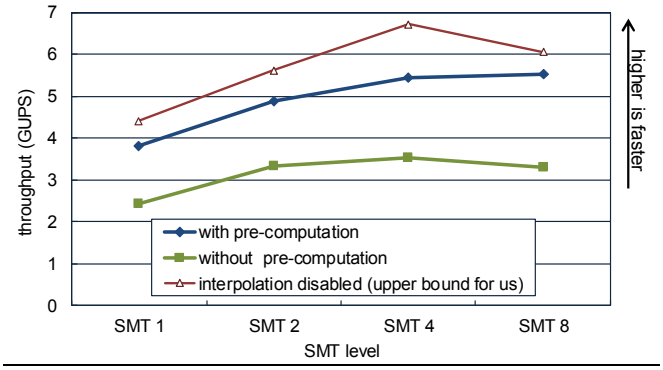

Figure 8. Performance (throughput (GUPS)) with and without our pre-computation technique for linear interpolation with different SMT levels on 5 cores of POWER8 for problem size of L = 512. SMT level means number of threads per core.

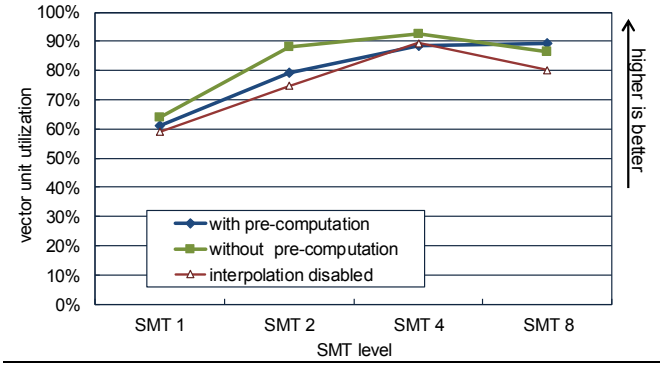

Figure 9. Vector unit utilization with and without our pre-computation technique for linear interpolation with different SMT levels on 5 cores of POWER8 for problem size of L = 512.

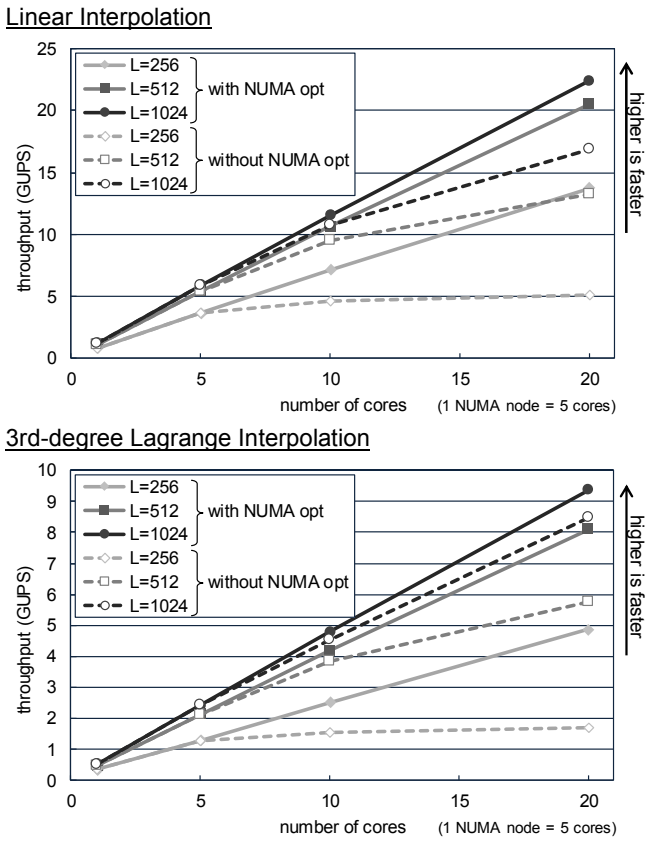

Figure 10. Performance scalability with and without optimization for NUMA on up to 20 cores (4 NUMA nodes) of POWER8. Since POWER8 chip has two dies on one socket, a system with two sockets has four NUMA nodes.

SMT1 (1 SMT thread per core) and 46% with SMT2. The best performance without the pre-computation was achieved with SMT4, and using SMT8 slightly degraded the performance due to increased cache misses caused by a larger memory footprint. With the pre-computation, we obtained the best result with SMT8, and SMT4 was the close second best.

The improvements in vector unit utilization with increasing SMT level was most significant when increasing the SMT level from 1 to 2 regardless of the use of the pre-computation. Since the numbers of executed vector instructions were almost constant when changing the SMT level, the changes in the vector unit utilization caused by the SMT level shown in Figure 8 were correlated with the changes in the execution time. Hence, the performance improvements were also most significant from SMT1 to SMT2. Improvements from using more SMT threads over SMT2 were relatively small compared to the changes between SMT1 and SMT2. For the 3rd-degree Lagrange interpolation, these trends were mostly similar. These results show that our pre-computation technique can improve the performance of this workload even on systems with fewer SMT threads per core.

Figure 10 illustrates the performance scalability with and without the optimization for the NUMA architecture described in Section III.A. With the NUMA optimization, the performances of both interpolation algorithms scaled well, even with 20 cores (2 sockets, 4 NUMA nodes). The increase in speed over the single core execution was up to 18.9x and 18.7x by 20 cores (for L = 1024) for the bilinear and Lagrange interpolations, respectively. However, the scalability was much poorer without the NUMA optimization when we used multiple NUMA nodes due to the overhead of remote memory accesses. The performance improvements caused by the NUMA optimization were more significant for the bilinear interpolation compared to the 3rd-degree Lagrange interpolation. The number of total memory accesses were not that different for the two algorithms as shown in Figure 7, but the execution time was much longer for the 3rd-degree Lagrange interpolation due to higher computation cost. Hence the higher communication-and-computation ratio of the bilinear interpolation resulted in more significant gains caused by the NUMA optimization.

IV. Related Work

In many applications, pre-processing of input data is a common technique to improve the end-to-end performance of the workload; hence, a variety of pre-processing techniques have been studied. For example, pre-conditioning techniques to improve memory access locality and vector utilization have been widely studied for sparse matrix systems. However,

there is no known pre-processing technique to make the grid-based interpolation more efficient. Also, the costs of the existing pre-processing techniques are often high; hence, the use of such pre-processing techniques may be justified only when the input data is repeatedly used. In the case of the pre-conditioning for sparse matrix systems, for example, a pre-conditioned matrix is often repeatedly used in an iterative method such as the conjugate gradient method; hence, the cost of pre-conditioning is amortized. Unlike such pre-processing techniques, our pre-computation technique incurs a much smaller overhead compared to the benefit we can obtain in one execution of the reconstruction phase. Hence, we can use the pre-computation technique even if each projection image is processed only once.

Due to its importance, the 3D cone-beam CT reconstruction has been studied for a long time. The FDK algorithm by Feldkamp *et al.* [3] is one of the most popular reconstruction algorithms. The FDK algorithm and its variants have been implemented on various hardware platforms including general-purpose processors [4, 8, 9], FPGAs [10], Xeon Phi [4], Cell BE processors [11], and GPUs [6, 12-13]. However, it was difficult to conduct fair comparisons of such implementations on different hardware with various optimization techniques involved in terms of the execution performance and the quality of reconstructed images. Recently, RabbitCT [2], an open framework for benchmarking the 3D cone-beam CT reconstruction, has provided a way to fairly compare algorithms and implementations by providing a benchmark framework and also high-quality input images and reference results.

Although our baseline implementation follows FastRabbit [4, 9], a state-of-the-art implementation for Intel's processors, our implementation outperformed FastRabbit running on a 2-socket machine (Intel IvyBridge) [4] by 2.9x using the same number of cores as for L = 512. Although this higher performance is partially due to hardware differences (with a higher frequency and larger number of SMT threads, but without 256-bit vector instructions), our technique plays a critical role in exhibiting superior performance, as shown in Figure 4. FastRabbit uses a naive formula to calculate the bilinear interpolations like our baseline implementation.

Although we implemented our pre-computation technique only on POWER8 processors, our technique can be applied for other processors. In RabbitCT benchmark, which uses bilinear interpolation, GPUs achieved much higher throughput compared to general-purpose CPUs such as POWER8 since GPUs support bilinear interpolation by hardware [6,12-13]. Because our pre-computation is not limited to bilinear interpolation, our technique is also beneficial for GPUs if a higher order interpolation algorithm, which is not supported by GPU hardware, is used to achieve higher image quality. Real-world CT systems often use higher order interpolation algorithms.

V. CONCLUSION

We developed a technique to accelerate interpolation of grid data (2D projection images) at a non-grid point. We argued that our pre-computation technique significantly improves the throughput of a medical imaging workload without degrading the accuracy; it exhibited up to 75% and 57% speed ups in the RabbitCT benchmark for bilinear and 3rd-degree Lagrange interpolation algorithms respectively. Since the interpolation of grid data to obtain data at a non-grid point is a common operation in many workloads, our new technique can contribute to a wider range of workloads and algorithms than just the 3D CT reconstruction we described in this paper.

REFERENCES

[1] W. C. Scarfe and A. G. Farman, What is cone-beam CT and how does it work?, Dental Clinics of North America, vol 52, pp. 707–730 (2008).

[2] C. Rohkohl, B. Keck, H. G. Hofmann, and J. Hornegger, RabbitCT – An Open Platform for Benchmarking 3-D Cone-beam Reconstruction Algorithms, *Medical Physics*, vol. 36, pp. 3940-3944 (2009).

[3] L. Feldkamp, L. Davis, and J. Kress, Practical Cone-Beam Algorithm, *J.ournal of the Optical Society of America*, vol. A1, no. 6, pp. 612-619 (1984).

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

[5] Q. Wang and K. D. Squires, Large eddy simulation of particle deposition in a vertical turbulent channel flow, *International Journal of Multiphase Flow*, Vol. 22, No. 4, pp. 667-683 (1996).

[6] E. Papenhausen, Z. Zheng, K. Mueller, GPU-accelerated back-projection revisited: squeezing performance by careful tuning, In *Workshop on High Performance Image Reconstruction* (2011).

[7] H. Inoue, Efficient Tomographic Reconstruction For Commodity Processors with Limited Memory Bandwidth, In *Proceedings of IEEE International Symposium on Biomedical Imaging* (2016).

[8] M. Gschwind and W. Schmidt, Bridging endian-dependent SIMD vector representation with compiler optimization, In *Workshop on Programming models for SIMD/Vector processing* (2015).

[9] J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger, and G. Wellein, Pushing the limits for medical image reconstruction on recent standard multicore processors, *Int. J. High Perform. Comput. Appl*. 27, 2, pp. 162-177 (2013).

[10] B. Heigl and M. Kowarschik, High-Speed Reconstruction for C-Arm Computed Tomography, In *Proceedings of Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, pp. 25-28 (2007) .

[11] H. Scherla, M. Koernera, H. Hofmanna, W. Eckertb, M. Kowarschikc, and J. Horneggera, Implementation of the FDK Algorithm for Cone-Beam CT on the Cell Broadband Engine Architecture, In *Proceedings of SPIE Proceedings Vol. 6510 Medical Imaging: Physics of Medical Imaging* (2007).

[12] E. Papenhausen and K. Mueller, Rapid rabbit: Highly optimized GPU accelerated cone-beam CT reconstruction, In *Proceedings of the Nuclear Science Symposium and Medical Imaging Conference*, pp. 1-2 (2013).

[13] T. Zinßer and B. Keck, Systematic Performance Optimization of Cone-Beam Back-Projection on the Kepler Architecture, In *Proceedings of Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, pp. 225-228 (2013).