Accelerating Tomographic Reconstruction with Efficient Interpolation

Hiroshi Inoue
IBM Research – Tokyo, 19-21 Nihonbashi Hakozaki-cho, Tokyo, 103-8510, Japan
University of Tokyo, 7-3-1 Hongo, Tokyo, 113-0033, Japan
inouehrs@jp.ibm.com

***Abstract*—The backprojection operation is a core building block of a variety of medical imaging applications such as a three-dimensional (3D) cone-beam computed tomography (CT) reconstruction. The 3D cone-beam CT reconstructs a structure in a 3D volume by backprojecting multiple projection images from different angles into the 3D volume. The intensity value at a projected (non-grid) point on a 2D image is obtained by interpolating values at the neighboring grid points. This paper proposes our new technique to efficiently execute the interpolation operation with SIMD instructions. The key to high performance is that this technique 1) reduces the amount of computation for the interpolation operation and 2) avoids scattered memory accesses that offset the benefit of SIMD instructions by creating a pre-computed table from the projection image. Our pre-computation technique yields up to 75% performance improvements in the RabbitCT benchmark on POWER8 processors. Our implementation achieves about 2.9x higher performance than the previous best score of RabbitCT on general-purpose processors using the same number of cores. Our implementation can process more than 60 projection images per second for the most common resolution of 512^3 using only one socket.**

Keywords—Backprojection, 3D cone-beam CT, SIMD, Interpolation

I. INTRODUCTION

Three-dimensional (3D) computed tomography (CT) [1] is one of the key components of many clinical workflows. It reconstructs a structure in a 3D volume by backprojecting multiple projection images from different angles into the 3D volume. The backprojection is a compute-intensive operation and known as the most time consuming part of the CT workload. Figure 1 shows a schematic overview of a 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 3D cone-beam CT. RabbitCT allows algorithm developers to compare the execution performance and accuracy of their backprojection algorithms. It provides high-quality pre-processed projection images of a rabbit captured by a C-arm CT system, a benchmark driver for executing the performance test and a reference implementation of the FDK algorithm [3], a widely used non-iterative reconstruction algorithm. For each projection image, each volume element (voxel) in the 3D volume is projected onto the projection image based on the transformation matrix (*projection matrix*), then the value at the projected point is obtained by performing a bilinear interpolation of the neighboring four pixels. The interpolation is critically important for image quality but causes significant overheads with SIMD (single instruction, multiple data) instructions of today's processors [4]. One reason is the large number of arithmetic instructions to calculate the interpolated value. Another reason is reading values from the four pixels requires scattered memory accesses, which reduces the efficiency of vector instructions, even with the hardware gather/scatter support of the latest processors.

In this paper, we describe our new technique to accelerate the interpolation operation; hence, the entire CT workload, by significantly reducing the number of arithmetic instructions required and also by avoiding scattered memory accesses. We create a pre-computed table for each input projection image in memory, and the backprojection loop accesses the pre-computed table instead of the projection image. For realistic resolutions, the benefit of using the pre-computed table in the backprojection phase is much larger than the overhead of pre-computation. This technique improves the performance of RabbitCT benchmark by up to 75%. We implemented this new technique on 2-socket POWER8 processors, and it outperformed the best score previously reported on a machine with 2-socket general-purpose processors [4] by 2.9x using the same number of cores.

The basic idea of our pre-computation technique is applicable for a much wider range of applications, algorithms, and hardware over those we evaluate in this paper. Although we evaluated our technique to accelerate the bilinear interpolation in RabbitCT, which implements a non-iterative reconstruction algorithm, it can be applicable for higher-order interpolation algorithms and for other imaging applications. For example, some iterative tomographic-reconstruction and image-registration algorithms are good targets for our technique because they also execute interpolation frequently like non-iterative reconstruction algorithms. Furthermore, certain non-imaging stencil applications are also potential targets. For instance, particle-in-cell simulations of two-phase flows, such as blood flow simulations, heavily execute the interpolation from surrounding grid (mesh) points to obtain the flow parameters at the locations of the particles; hence, they

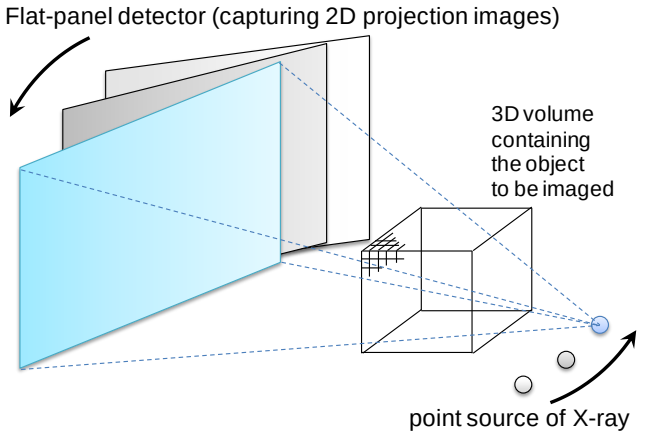

Figure 1. Overview of 3D cone-beam CT. A flat-panel detector and an X-ray source move around the 3D object to be imaged to capture 2D projection images from different angles.

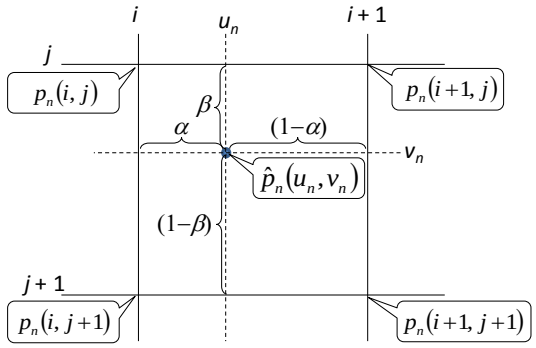

Figure 2. Overview of the projected point $\{u_n, v_n\}$ and neighboring four grid points.

are interesting targets for our pre-computation technique. Our technique can be implemented on GPUs, which can potentially provide much higher peak throughput.

Our current implementation can process more than 60 projection images per second for the most common problem size of 512^3 using only one socket (10 cores) POWER8 processor. Hence, it is realistic to use general-purpose processors for real-time reconstruction of a 3D volume from imaging devices capable of 60-fps image acquisition without using additional accelerators such as GPUs [5, 6] or FPGAs [7].

The rest of the paper is organized as follows. In Section 2, our pre-computation technique is described. In Section 3, our implementation and results are explained. Section 4 discusses related work. Finally, concluding remarks are given in Section 5.

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. We denote n-th projection image I_n. For each of the projection images, the reconstruction algorithm projects all voxels onto the I_n and updates the density value for the voxel based on the intensity value at the projected point obtained by the bilinear interpolation. 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 the four problem sizes, L=512 is the most common and L=128 is a toy benchmark mostly for debugging (hence 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 the point $\{u_n, v_n\}$ in the I_n as follow. Here, a_n is a projection matrix determined by the system geometry and provided for each I_n.

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

The 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 $p_n(i, j)$ at the neighboring four pixels.

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

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

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

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

In addition to the execution time, RabbitCT reports on the errors from the reference implementation to evaluate the accuracy of the reconstruction algorithm.

B. *Our Pre-computation Technique*

The reconstruction algorithm described in Section II.A has inherently huge parallelism; hence, we can accelerate the execution of the algorithm by exploiting the two types of parallelisms available in processors: thread-level parallelism by multiple cores and data parallelism by SIMD instructions.

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 |  |  |  |  x = Ix × R_L + O_L
14 |  |  |  |  y = Iy × R_L + O_L
15 |  |  |  |  z = Iz × R_L + O_L
16 |  |  |  |  calculate p̂_n by equation 4
17 |  |  |  |  update density of the voxel by p̂_n / w_n^2
18 |  |  |  end
19 |  |  end
20 |  end
21 end
```

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

without our pre-computation

```
1  for each projection image I_n (n = 0 to N - 1)
2  |  // create zero-padded copy [1]
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 III.A for detail)
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 |  |  |  |  x = Ix × R_L + O_L
14 |  |  |  |  y = Iy × R_L + O_L
15 |  |  |  |  z = Iz × R_L + O_L
16 |  |  |  |  calculate p̂_n by equation 2
17 |  |  |  |  update density of the voxel by p̂_n / w_n^2
18 |  |  |  end
19 |  |  end
20 |  end
21 end
```

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

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

Efficiently exploiting these parallelisms is critical to achieve good 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. The inefficiency is due to the scattered memory accesses to read the I_n from four pixels. A SIMD load instruction can load multiple elements only when they are contiguous in memory; hence, noncontiguous memory access incurs overhead of executing multiple load instructions and mixing the loaded elements. Although some latest processors, such as Intel Haswell, support gather and scatter memory access instructions to load or store noncontiguous memory, they are still slower than contiguous memory accesses.

To efficiently vectorize Equation 2, we modify it 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

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

Because these coefficients are independent from u_n and v_n (hence independent from x, y, and z), we can pre-compute these values before executing the reconstruction and store the results in an in-memory pre-computed table. By storing C_0 to C_3 for each pixel contiguously in memory aligned on 128-bit address boundaries, we can totally avoid scattered or unaligned memory accesses for the I_n because, as shown in Equation 4, these four values are enough to compute $\hat{p}_n(u_n, v_n)$, and we do not need to access the I_n after the pre-computation phase. Each iteration of the inner-most loop of our algorithm loads C_0 to C_3 for four pixels 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 pixels in one vector register.

Another benefit of using the pre-computed table is that Equation 4 is much more efficient to compute than Equation 2. Equation 4 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 twelve instructions to compute in our implementation. The multiply-and-add (a.k.a. FMA on x86 processors) instruction is supported on many high-performance

processors. Figure 3 shows the pseudocode with and without our technique.

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 to single precision just before storing in the pre-computed table. When we use the single-precision numbers during the pre-computation, we observe non-negligible degradation in accuracy. As discussed later, this degradation is not significant when we use double-precision numbers in the pre-computation phase.

Our pre-computation technique uses additional memory space for the pre-computed table. The size of the table is four times as large as an I_n because we hold four values, $C_0(i, j)$ to $C_3(i, j)$, per pixel. However, this memory overhead is not significant compared to the sizes of the other data structures such as the density values for voxels. Also, the access to the pre-computed table causes more cache misses due to its larger footprint than the I_n. However, the benefit of reduced computations was much more significant than the drawback of increased cache misses.

The pre-computation phase consumes additional computation time. As we experimentally show later, the overhead in computation time 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. We expect that the overhead of the pre-computation will become more insignificant on future CT systems, which employ higher resolution in both 2D projection image and 3D volume, because the execution time of the pre-computation phase is proportional to the square of the resolution while the reconstruction phase follows the cube of the 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 for other general-purpose processors such as Intel Xeon.

A. Implementation

We implemented the reconstruction algorithms in C++ and vectorized them by using SIMD intrinsics provided by IBM XL C++ compiler v13.1. We used single-precision floating point numbers in the implementation unless we explicitly denote 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, 8], a state-of-the-art implementation for Intel's processors. The baseline optimizations include 1) replacing divide instructions by a reciprocal estimate instruction, 2) skipping the voxels that cannot be projected onto the 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 the I_n with 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. The 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.

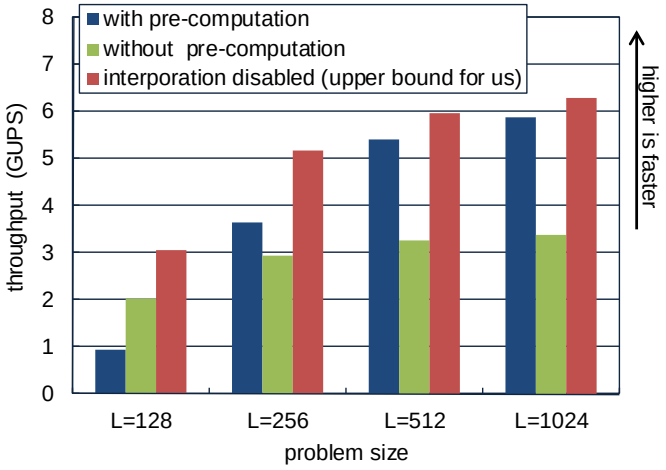

Figure 4. Performance (throughput in GUPS) with and without our pre-computation technique on 5 cores of POWER8.

Table 1. Accuracy (root mean squared error in HU) with and without our pre-computation technique on 5 cores of POWER8. Our pre-computation technique exhibits high performance similar to upper bound (interpolation disabled) without significant loss in accuracy.

Root Mean Squared Error (HU)

Problem size	With pre-computation	Without pre-computation	Interpolation disabled
L=128	0.533	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

Many recent multi-socket processors 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. Then each NUMA node processes different projection images rather than multiple NUMA nodes cooperating on one image. After processing all the projection images, we calculate the final density value for each voxel by summing up the partial results from all NUMA

Table 2. Execution time breakdown on 5 cores of POWER8

Problem size	With pre-computation			Without pre-computation	Without interpolation
	precomputation	reconstruction	total		
L=128	0.94 (46%)	1.11	2.06	0.97	0.64
L=256	0.94 (22%)	3.36	4.30	5.31	3.03
L=512	0.94 (4.0%)	22.20	23.13	38.39	21.00
L=1024	0.94 (0.6%)	169.30	170.23	297.41	159.4

- The numbers show the execution time per projection image (in msec).
- The percentages shown in parenthesis show the ratios to the total execution time.

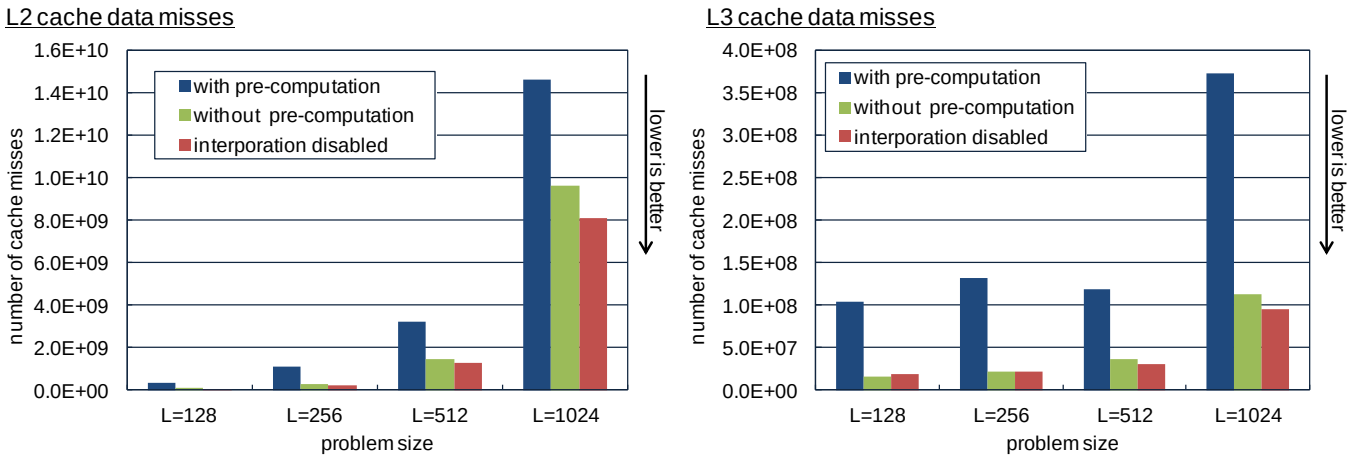

Figure 5. L2 and L3 cache data misses with and without pre-computation with 5 cores.

nodes. 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 values for these values. An atomic operation (atomic add) is used only when a thread picks a next block from a 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 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 8-way SMT. Each core equipped with 256-KB L2 cache and 8-MB L3 cache memory. We disabled the dynamic frequency scaling for more consistent and reproducible results. We measured the performance 16 times and explain the average results.

Figure 4 and Table 1 illustrates the performance (throughput in GUPS, Giga voxels updates per second) and accuracy (the root mean squared error from the reference result in HU, *Hounsfield unit*) for 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 (40 threads). It also illustrates the performance when we totally disabled the bilinear 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 a bilinear interpolation. Our pre-computation technique exhibited up to 75% (for L=1024) performance improvement over the baseline (without pre-computation) without significant losses in accuracy. There was only a 6.8% performance gap against the upper bound (when the bilinear interpolation was disabled). When the interpolation was disabled, unlike with our technique, 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 used single-precision numbers, the accuracy would become as poor as about 2.75 HU.

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 I_n size, e.g., L=128 in Figure 4, the pre-computation degrades performance. Table 2 shows the execution time breakdown 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 too small.

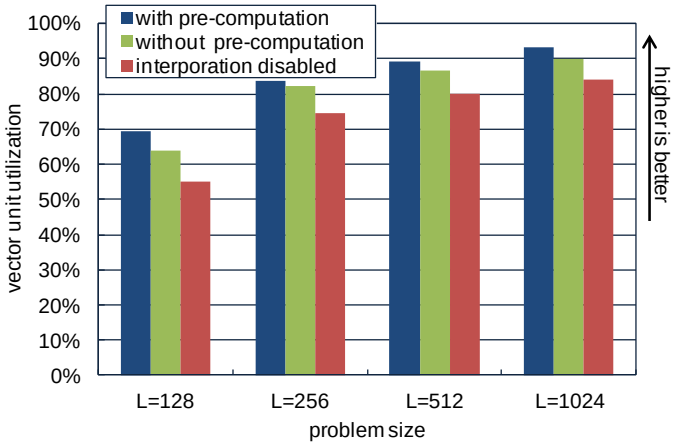

Figure 6. The vector unit utilization with and without our pre-computation technique on 5 cores of POWER8. Note that one POWER8 core can execute two vector instructions per cycle.

For more insight into the improvements with our pre-computation technique, Figure 5 compares the numbers of L2 and L3 cache data misses as measured by the performance monitor of the processor. Our technique increased the cache misses in both L2 and L3 cache regardless of the problem size as a trade-off for reducing the number of executed SIMD instructions and avoiding scattered memory accesses. The pre-computed table is 4x as large as the original 2D image; hence, the larger memory footprint of the pre-computed table increased cache misses.

Figure 6 shows the vector unit utilization. The utilization becomes 100% when one core executes two vector instructions every cycle. As shown in Figure 6, 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 bottle neck in this workload even with the increased cache misses with the pre-computation, and hence the performance gain due to the reduced SIMD instructions is much more significant than the overhead due to the increased cache misses.

The POWER8 processor we used in the experiments supports 8 SMT threads per core, while other general-purpose processors typically support smaller number of SMT threads; e.g. latest Intel x86 processors support 2-way SMT by HyperThreading. Using higher SMT level can potentially hide the cache miss latency and improve the vector unit utilization. To confirm that our pre-computation is effective on other processors that only supports smaller number of SMT threads per core, we show the performance and the vector unit utilization for configurations with 1 to 8 SMT threads per core on POWER8 in Figure 7 and 8. Our pre-computation improved the performance regardless of the SMT level. The performance gain by the pre-computation was 57% with SMT1 (1 SMT thread per core) and 46% with SMT2. When we do not use the pre-computation, there were not significant performance gain from using more than 2 SMT threads per core. 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 larger memory footprint. With the pre-computation, we got the best result with SMT8 and SMT4 was the close second best.

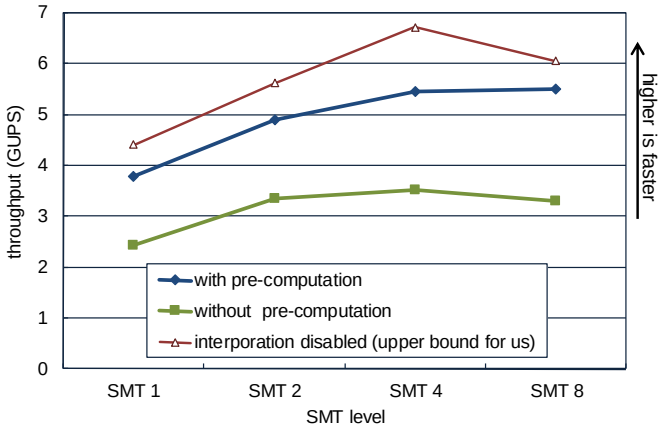

Figure 7. Performance (throughput in GUPS) with and without our pre-computation technique with different SMT levels on 5 cores of POWER8 for the problem size of L=512. Here, SMT level means the number of SMT threads per code.

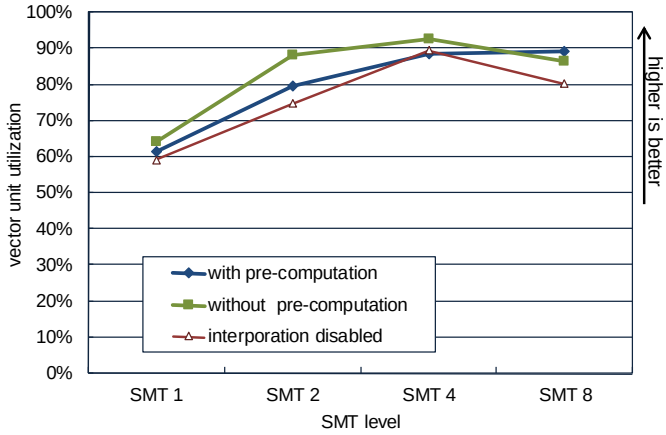

Figure 8. The vector unit utilization with and without our pre-computation technique with different SMT levels on 5 cores of POWER8 for the problem size of L=512.

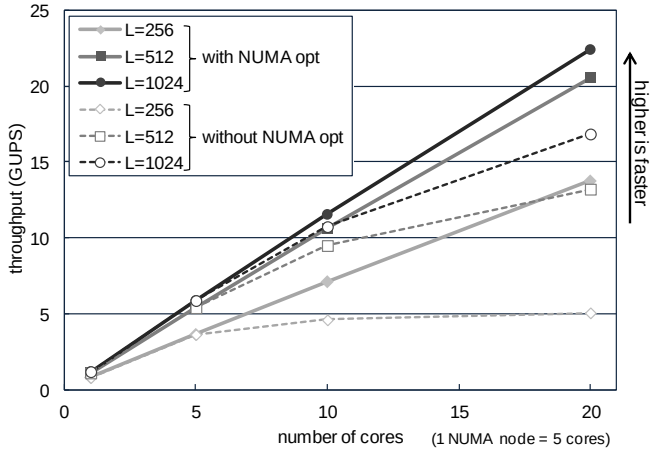

Figure 9. Performance scalability with and without optimization for NUMA on up to 20 cores (4 NUMA nodes) of POWER8. Since POWER8 chip has two dies on one socket, a system with two sockets has four NUMA nodes.

Table 3. Summary of RabbitCT Performance of various implementations (L = 512)

Processor	# Core / # Boards	Year	Source	GUPS
POWER8 3.69 GHz	20 cores (2 sockets)	2015	Ours	20.5
POWER8 3.69 GHz	10 cores (1 socket)	2015	Ours	10.6
IvyBridge-EP 2.2 GHz	20 cores (2 sockets)	2014	Paper [4]	about 7.0
Westmere-EX 2.4 GHz	40 cores (4 sockets)	2011	Official ranking	8.3
Xeon Phi 5110P	1 board	2014	Paper [4]	about 8.5
nVidia GTX 670	2 boards	2014	Official ranking	152.9

The vector unit utilization shown in Figure 8 was almost determined by the performance since the number of executed vector instructions were almost constant when changing the SMT level. The improvements in vector unit utilization with SMT was most significant when increasing the SMT level from 1 to 2. Improvements from using more SMT threads over SMT2 were relatively small compared to the changes between SMT1 and SMT2. These results show that our pre-computation technique can improve the performance of this workload even on systems with smaller SMT threads per core.

Figure 9 illustrates the performance scalability with and without the optimization for the NUMA architecture described in Section 3.1. With the NUMA optimization, the performance scales well, even with 20 cores (2 sockets, 4 NUMA nodes); the increase in speed over the single core execution was up to 18.9x by 20 cores (for L=1024). However, the scalability was much poorer without the NUMA optimization when we used multiple NUMA nodes due to the overhead of remote memory accesses.

IV. Related Work

Due to its importance, the 3D cone-beam CT reconstruction has been studied for a long time. FDK algorithm by Feldkamp *et al.* [3] is one of the most popular reconstruction algorithm. FDK algorithm and its variants have been implemented on various hardware platforms including general-purpose processors [4, 8], FPGAs [7], Xeon Phi [4], Cell BE processors [9] and GPUs [5, 6]. However, it was not easy 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, gives a way to fairly compare algorithms and implementations by providing a benchmark framework and also a high-quality input images and the reference results.

By using our pre-computation technique, our implementation exhibited significantly higher performance than the previous results of the RabbitCT benchmark on general-purpose processors. Table 3 compares the performance of various implementations for the problem size of L=512. Our implementation outperformed the previous best score on a 2-socket machine (Intel IvyBridge) [4] by 2.9x using the same number of cores. Although this higher performance is partially due to hardware differences (with 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 Figures 4.

Despite the huge improvements in performance on general-purpose processors with our pre-computation, our implementation still falls behind those running on GPUs [4, 5]. GPUs support a bilinear interpolation in hardware in addition to the huge peak vector processing capability; hence general-purpose processors cannot compete with the GPUs in this specific workload even with our pre-computation technique. Because our pre-computation is not limited to bilinear interpolation, which is supported by hardware on GPUs, our technique is also beneficial for GPUs if a higher-order interpolation algorithm is used to achieve higher image quality.

V. Conclusion

We developed a technique to accelerate 3D volume reconstruction with the backprojection by enabling efficient interpolation on 2D projection images. We argued that our pre-computation technique significantly improves the throughput without degrading the accuracy by reducing the number of arithmetic instructions to calculate a bilinear interpolation while avoiding scattered memory accesses. The implementation of our technique exhibited about 2.9x higher performance than the previous best score of the RabbitCT benchmark running on general-purpose processors. Due to recent advances in general-purpose processors with increased number of cores and widening vector hardware, it is becoming more practical to use commodity processors instead of specialized hardware in a wider range of medical imaging applications.

References

[1] W. C. Scarfe, 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, *Journal 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 *Proceedings of Workshop on Programming models for SIMD/Vector processing* (2014).

[5] 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)

[6] 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)

[7] 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)

[8] 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)

[9] 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)