EFFICIENT TOMOGRAPHIC RECONSTRUCTION FOR COMMODITY PROCESSORS WITH LIMITED MEMORY BANDWIDTH

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

ABSTRACT

Three-dimensional (3D) computed tomography (CT) is one of the key components of many clinical workflows. Because CT reconstruction has been known as a compute-intensive workload, accelerating this workload using special-purpose accelerators, such as GPUs and FPGAs, or multi-socket server-grade processors has been widely studied. Due to recent advances in semiconductor technologies, even commodity processors, such as those used in PCs, can provide sufficient computing power for CT reconstruction by multiple cores with vector processing units. Despite their huge computing power, commodity processors often provide limited system memory bandwidth compared to server-grade processors due to constraints in cost and energy consumption. In this paper, we describe our memory-optimization technique and its implementation targeting on general-purpose processors with limited memory bandwidth. By reducing the memory-bandwidth requirement with batch processing, the memory optimization achieved up to 80% performance improvements in RabbitCT, a widely-used CT benchmark, on a quad-core processor with limited memory bandwidth. Without the memory optimization, the performance did not scale with more than two cores. The implementation can process about 40 projection images per second for the most common problem size of 512^3 with only four cores used. It is therefore practical to use such commodity processors in real CT systems without additional accelerators, which trade greatly increased cost and energy consumption for higher throughput.

Index Terms**—**CT Reconstruction, Memory bandwidth, Back-projection

1. INTRODUCTION

Three-dimensional (3D) computed-tomography (CT) reconstruction is known as a compute-intensive workload, and its performance has a significant effect on many clinical workflows. Due to its importance, many existing CT implementations exploit huge computing power of multi-socket server-grade processors [1, 2], such as Intel's Xeon, and accelerators including GPUs [3, 4] and FPGAs [5].

Because of today's advanced semiconductor technologies, commodity general-purpose processors, like those used in PCs or tablets, have multiple cores and vector (SIMD) processing units. Moreover, many commodity processors integrate GPU cores to accelerate processing of compute-intensive workloads. Hence, it is becoming increasingly realistic to use commodity general-purpose processors for 3D CT reconstruction. In contrast to their huge computing power, however, today's commodity processors typically provide only limited system memory bandwidth and cache size compared to multi-socket server-grade processors due to constraints on cost and energy consumption. To provide wider memory bandwidth, CT systems need to be equipped with more memory controllers in processors and more DIMM sockets on the system board; consequently, cost, system size, and energy consumption are all increased. On such commodity processors, as a result, the computation performance of the processor cores cannot be fully utilized due to a bottleneck in system memory bandwidth; that is, the performance does not scale linearly with increasing number of cores.

In this study, which targets commodity processors with limited memory bandwidth, we describe a method of memory optimization for the 3D CT reconstruction. The memory optimization was implemented on RabbitCT [6], an open benchmarking framework for benchmarking 3D cone-beam CT-reconstruction algorithms. Our memory optimization, which processes multiple projection images at once, drastically reduced memory-bandwidth requirement and enabled almost-linear performance scalability with increasing number of cores in a system with limited memory bandwidth, while the scalability was much poorer without the optimization. The performance gain achieved by the memory optimization was up to 80% when using four cores.

The implemented memory optimization can achieve 3.7x better performance per core compared to the previous best score on server-grade general-purpose processors [1]; in other words, using only four cores, it processes about 40 projection images per second for the most-common problem size of 512^3. Hence, it makes it possible to use commodity processors for 3D CT reconstruction without using additional accelerators such as GPUs.

2. METHODS

2.1. Baseline reconstruction algorithm [6]

RabbitCT evaluates the performance and accuracy of the back-projection operation for 3D volume reconstruction. It provides 496 projection images, whose size $S_x \times S_y$ is 1248×960 pixels, with a projection matrix for each image. The n-th projection image is denoted as I_n. For each projection image, the reconstruction algorithm projects all voxels onto I_n and updates the density value of the voxel based on the intensity value at the projected point obtained by 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 these sizes, L=512 is the most common one. Although different numbers of voxels are used, the same I_n is 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 I_n as follow. Here, a_n is a projection matrix determined by the system geometry and provided for each I_n.

$$u_n(x,y,z) = (a_0 x + a_3 y + a_6 z + a_9)/w_n(x,y,z),$$
$$v_n(x,y,z) = (a_1 x + a_4 y + a_7 z + a_{10})/w_n(x,y,z),$$
$$w_n(x,y,z) = a_2 x + a_5 y + a_8 z + a_{11}.$$

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

$$\hat{p}_n(u_n, v_n) = (1-\alpha)(1-\beta)p_n(i,j) + \alpha(1-\beta)p_n(i+1,j) + (1-\alpha)\beta p_n(i,j+1) + \alpha\beta p_n(i+1,j+1),$$
$$i = \lfloor u_n \rfloor,\ j = \lfloor v_n \rfloor,\ \alpha = u_n - \lfloor u_n \rfloor,\ \beta = v_n - \lfloor v_n \rfloor.$$

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

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

A pseudocode of the baseline reconstruction algorithm is shown in Figure 1(a). To accelerate the reconstruction, we preprocessed each projection image and stored the result in an in-memory pre-computed table. The reconstruction algorithm is implemented using SIMD instructions for both the pre-computation and reconstruction phases. Also, it employs the following three optimizations: (i) replacing divide instructions by reciprocal estimate instructions, (ii) skipping voxels that cannot be projected onto the projection image, and (iii) eliminating conditional branches for checking out-of-image-bound accesses by creating a copy of the projection image with zero padding around the image [1, 2].

(a) Naive reconstruction algoruthm: processing one image in each iteration (B=1)

```
for each projection image $I_n$
    // pre-computation phase
    for $i$ = 0 to $S_x$ - 1
        for $j$ = 0 to $S_y$ - 1
            pre-computation for image $I_n$
        end
    end
    // reconstruction phase
    for $Iz$ = 0 to $L$ - 1
        for $Iy$ = 0 to $L$ - 1
            determine range of $Ix$ to iterate
            for $Ix = Ix_{start}$ to $Ix_{end}$
                project and update voxel ($Ix$, $Iy$, $Iz$)
                for image $I_n$
            end
        end
    end
end
```

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

(b) Algorithm with memory optimization: processing multiple (B>1) images at once

```
for each batch of $B$ proj. images $I_n$ to $I_{n+B-1}$
    // pre-computation phase
    for $i$ = 0 to $S_x$ - 1
        for $j$ = 0 to $S_y$ - 1
            for $k$ = 0 to $B$ - 1
                pre-computation for image $I_{n+k}$
            end
        end
    end
    // reconstruction phase
    for $Iz$ = 0 to $L$ - 1
        for $Iy$ = 0 to $L$ - 1
            determine range of $Ix$ to iterate
            for $Ix = Ix_{start}$ to $Ix_{end}$
                for $k$ = 0 to $B$ - 1
                    project and update voxel ($Ix$, $Iy$, $Iz$)
                    for image $I_{n+k}$
                end
            end
        end
    end
end
```

Figure 1. Pseudocode of reconstruction algorithm without and with memory optimization by batch processing with batch size B.

2.2. Memory optimization technique

The minimum memory-bandwidth requirements for the baseline algorithm are considered as follows. For each projection image (each iteration in the outer-most loop in Figure 1), it is necessary to read the projection image itself and read and write the density values of voxels that are projected onto the current projection image. To assess the minimum memory-bandwidth requirement, it is assumed that all other accesses, such as those to the pre-computed table, do not cause cache misses and hence do not consume system memory bandwidth. The amount of data transfer for reading and writing the density values of voxels (e.g., 330 MB per projection image for L=512) is much larger than that for reading the projection image (4.7 MB) for realistic resolutions. Consequently, it is critical to reuse the density values of voxels within the cache memory of the processor to achieve good performance with very limited memory bandwidth.

A pseudocode of a CT reconstruction algorithm with our memory optimization is shown in Figure 1(b). To reduce the memory-bandwidth bottleneck, multiple projection images are processed at once. As the number of images to be processed at once (*batch size*) increases, the memory bandwidth used to transfer the density values of voxels is reduced. When batch size is B, the memory bandwidth required for transferring voxel data becomes only $1/B$ of the baseline algorithm since these values must be read and written only once per B images. In trade for reduced memory bandwidth required for voxel data, the memory footprint required for the pre-computed tables and projection images is increased by B times, potentially increasing cache misses. However, in total of on the whole, reduced required bandwidth for the voxel data and increased required bandwidth for the projection image made it possible to significantly reduce memory bandwidth (as empirically shown later).

Papenhausen *et al.* [3] employed a similar idea of processing multiple projection images per kernel invocation on GPUs to fully utilize the huge computation power of GPUs by reducing accesses to the global memory and also reducing the number of kernel invocations. Our results show that such memory optimization is important to achieve high performance on commodity processors with limited memory bandwidth, though their peak throughput is much lower than that of GPUs and no kernel invocation overhead is incurred.

3 EXPETIMENTAL RESULTS

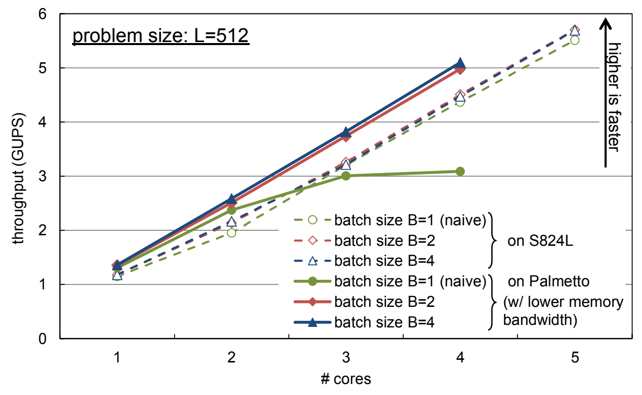
The proposed memory optimization was implemented and experimentally evaluated on a POWER8 processor. The evaluation results illustrate how the memory optimization affects the performance on two systems with different memory performances.

The memory optimization implemented was evaluated on two different systems with POWER8 processors. One system, called *S824L*, is a large server system with two 3.69-GHz POWER8 processor with 256 GB of system memory. It has 20 processor cores (10 cores per socket) in total. Another (smaller) system, called *Palmetto*, is equipped with a quad-core version of a 4.32-GHz POWER8 processor with 64 GB of system memory. The first system equips four memory controller chips (called *Centaur*) per five cores, while the latter equips only one memory controller chip per four cores [7]. Hence, there is about 4x gap in the memory bandwidth per computation performance (FLOPS) of two systems. A gap in the memory bandwidth also exists between Intel's server processors and client processors. For example, the Xeon E7 v3 series server-grade CPUs (Haswell-EX) provides 4x wider memory bandwidth per socket compared to the same generation CPUs for clients (4th generation Core i7).

The reconstruction algorithms were implemented in C++ and compiled by using IBM XL C++ compiler v13.1. We used single-precision floating point numbers in the implementation. Both systems ran Ubuntu Linux 14.10 LE. We disabled the dynamic frequency scaling for more consistent and reproducible results. For each data point, two configurations, four threads pre core and eight threads per core, were evaluated, and the better performance was reported because neither configuration consistently outperformed the another. We measured the performance 16 times and show the average results.

Figure 2 illustrates the throughput performance (in GUPS, Giga voxels updates per second) for two problem sizes, *L*=512 and *L*=1024, on the two systems with various batch size (*B*); i.e., *B* projection images were processed in each iteration of the inner-most loop. The reconstructed image quality was not affected by this optimization. For both problem sizes, the naive algorithm without batch processing (*B*=1 in the figure) did not scale beyond two cores on Palmetto due to a memory-bandwidth bottleneck. By processing multiple projection images at once, i.e., *B*=2 or *B*=4, the performance scaled almost linearly because the proposed optimization makes it possible to use the memory bandwidth more efficiently. When all four cores are used, the performance from batch size of 1 to 4 was improved by about 65% for *L*=512 and 80% for *L*=1024.

On S824L, on the contrary, the optimization only marginally improved performance by up to 4%. This was because S824L has a huge memory bandwidth, so memory bandwidth was not a major bottleneck, even when



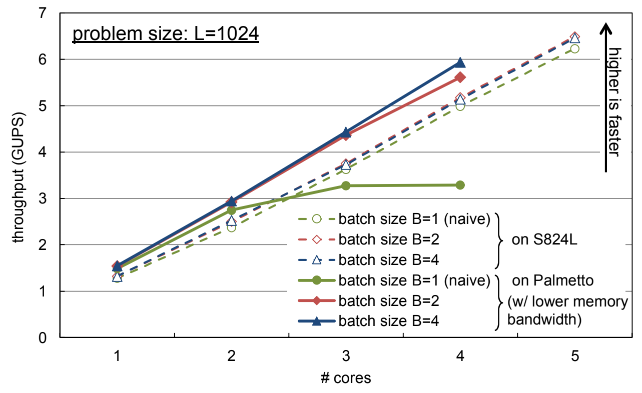


Figure 2. Performance with various batch size *S* on two systems (using 1 NUMA node). S824L has about 4x more memory bandwidth per processor's FLOPS than Palmetto.

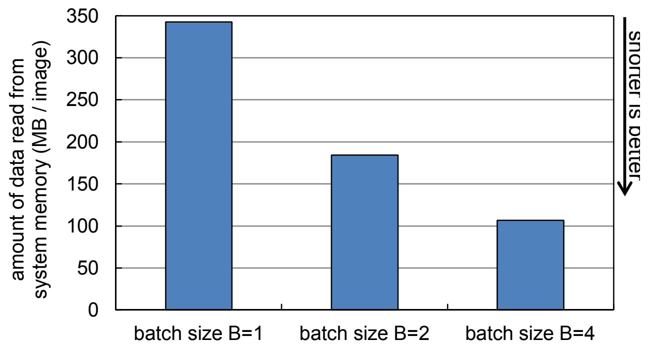


Figure 3. Amount of data read from system memory for L=512 with various batch sizes measured on Palmetto using four cores.

Table 1. Summary of performances of various implementations on RabbitCT ($L = 512$)

Category	Processor	# Core / # Boards	Year	Source	GUPS
Low memory bandwidth	**(Palmetto) POWER8 4.32 GHz**	4 cores (1 socket)	2015	Ours	5.1
Server-grade processor	**(S824L) POWER8 3.69 GHz**	20 cores (2 sockets)	2015	Ours	21.4
	(S824L) POWER8 3.69 GHz	10 cores (1 socket)	2015	Ours	10.8
	IvyBridge-EP 2.2 GHz	20 cores (2 sockets)	2014	Paper [1]	about 7.0
	Westmere-EX 2.4 GHz	40 cores (4 sockets)	2011	Official ranking	8.3
Accelerator	Xeon Phi 5110P	1 board	2014	Paper [1]	about 8.5
	nVidia GTX 670	2 boards	2014	Official ranking	152.9

projection images were processed one by one. On both systems, the differences in performance for batch sizes of 2 and 4 were quite small, though the batch size of 4 consistently showed slightly higher throughput than the batch size of 2.

To confirm that larger batch sizes really reduced the memory bandwidth requirements, we show the amount of data read from the system memory (as measured by using the processor's performance counter on Palmetto) in Figure 3. For this measurement, the hardware prefetcher of the processor was disabled to capture all memory accesses accurately with the performance counter. According to the figure, the amount of the system memory accesses was reduced in the case of larger batch size as expected. This reduction in memory bandwidth resulted in better scalability with our memory optimization on Palmetto shown in Figure 2.

By using the memory optimization by the batch processing described in this paper, our implementation exhibited significantly higher performance than the previously achieved results of the RabbitCT benchmark on general-purpose processors. Table 1 compares the performance of various implementations for problem size of L=512. Our current implementation on the Palmetto system achieved about 3.7x higher performance per core than the previous best score on a two-socket machine (Intel IvyBridge) [1]. Our implementation on S824L outperformed this previous best score by more than 3x using the same number of cores. The throughput on Palmetto was more than 5 GUPS using only four cores despite of its limited memory bandwidth. Although the total performance on Palmetto still falls behind that of implementations running on server-grade processors or GPUs, using the commodity processor yields much lower system costs and lower energy consumptions. These are both very important characteristics in real systems.

4. CONCLUSION

A memory optimization technique for accelerating 3D volume reconstruction with back-projection on processors with limited memory bandwidth, such as commodity processors for client systems, was developed. The implemented algorithm scaled well on a system with limited memory bandwidth per FLOPS as well as on a system with larger memory bandwidth. It exhibited performance of more than 5.0 GUPS on RabbitCT on a quad-core system with limited memory bandwidth. Due to recent advances in the computation power of commodity processors with increased number of cores and widening vector hardware, limited system memory performance is the largest obstacle to use commodity processors in a wider range of medical imaging applications. Our results presented in this study show that algorithms optimized for commodity processors with limited memory bandwidth make it more practical to use such commodity processors in real systems instead of costly server-grade processors or specialized hardware.

5. REFERENCES

1. 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 many-core chips, In *Proceedings of Workshop on Programming models for SIMD/Vector processing* (2014).
2. 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)
3. E. Papenhausen, Z. Zheng, K. Mueller: GPU-accelerated back-projection revisited: squeezing performance by careful tuning, In *Workshop on High Performance Image Reconstruction* (2011)
4. 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)
5. 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)
6. 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)
7. The Linley Group: *POWER8 Hits the Merchant Market*, (2014) accessed November 2015 at http://www-03.ibm.com/systems/power/advantages/smartpaper/memory-bandwidth.html