Critique of "MemXCT: memory-centric X-ray CT reconstruction with massive parallelization" by SCC Team from Georgia Tech

Nicole Prindle, Ali Kazmi, Aman Jain, Albert Chen, Marissa Sorkin, Sudhanshu Agarwal, Richard Vuduc, Vijay Thakkar

Abstract—For the 2020 Student Cluster Competition, we reproduced results from "MemXCT: Memory-Centric X-ray CT Reconstruction with Massive Parallelization" (Hidayetoğlu et al.). Reproducibility is of critical importance to the scientific community, not just to verify correctness of results but also to see how easily others can understand and work with the given methods. MemXCT is an approach for image reconstruction in X-ray ptychography, which has a broad range of applications in materials science. MemXCT is not the only X-ray tomography algorithm, though; as opposed to compute-centric algorithms, it is designed to scale better by optimizing for memory bandwidth and memory latency. MemXCT also applies several key optimizations in order to ease memory pressure. In this paper, we test the performance and strong scaling of MemXCT on 1 to 256 AMD CPU cores (1-4 nodes) and 1-16 Nvidia V100 GPUs (1-4 nodes). We confirm the impact of MemXCT's optimizations. Still, we find that the performance of some important loops in the MemXCT kernel is much lower on the AMD processors (with AVX2) of our CPU nodes compared to the Intel CPUs (with AVX-512) used in the original paper. We also confirm MemXCT performance on Tesla V100 GPUs, as reported in the paper.

Index Terms—Reproducible computation, Parallel Computing, Student Cluster Competition, MemXCT

1 Introduction

The algorithm introduced by the MemXCT [1] paper is notable in that it minimizes memory and latency overhead on application performance. It memoizes ray tracing, partitioning the domain better between parallel units. At its core, MemXCT consists of SpMV kernels; one of the key kernel changes is transforming a scatter operation to a gather operation. However, this leads to increased indirect accesses. MemXCT therefore applies several optimizations in order to help cache performance and minimize memory pressure from indirect accesses; it performs a pseudo-Hilbert ordering of the input domain for better cache locality, and uses multi-stage buffering to pre-load indirect accesses into L1 cache. We attempt to reproduce the results from the paper on both CPUs and GPUs. As we are using AMD CPUs without an AVX-512 instruction set, we will simultaneously be confirming the importance of these optimizations to MemXCT's performance. By performing strong scaling experiments on multiple CPU and GPU nodes, we attempt to confirm that MemXCT scales easily to larger clusters as the problem size per device decreases.

2 Experimental Setup

Experimental runs were performed on Microsoft Azure, with a setup that differed from the original paper in several key ways. The compute nodes used for CPU benchmarks are Azure's HB120rs_v2 nodes, which are dual socket nodes containing AMD EPYC 7V12 64-core processors [2], with 200 Gb/s HDR InfiniBand. Each node has 480 GB of memory, with a theoretical bandwidth of 350 GB/s. The compute nodes used for GPU benchmarks are Azure's NC24rs_v3 nodes, each having 24 cores and 4 Nvidia Tesla V100 GPUs [3]. The master node of our GPU cluster is Azure's D64ds_v4 node, for compatibility with the compute nodes [4].

We used an Azure CentOS-HPC image, specifically `OpenLogic:CentOS-HPC:7.7:7.7.2020062600`, as our operating system. For compilers, we used GCC 9.3.1, which is supplied by `devtoolset-9-gcc` from the CentOS Software Collection. We used the OpenMPI 4.0.4 provided by HPC-X v2.7.0 in the image as an MPI library, which is compatibile with InfiniBand. For GPU code, we use CUDA 11.1.1, which is installed directly from Nvidia's website as an RPM package. Further information regarding compilation flags, environment variables, and other build details can be found in our corresponding repository (see section 7). Other than libraries, minimal modifications were made. Our primary change was fixing an error in the GPU code, which did not assign device numbers properly when run across multiple nodes having multiple GPUs each.

We used three datasets viz. CDS1 (small), CDS2 (medium), and CDS3 (large) as provided in the competition datasets.

3 Description of experimental run

For the experiment, we implemented separate clusters for CPU and GPU runs. On each of the clusters, we carried out two sets of experiments: single-device runs and multi-node runs. The performance from the MemXCT paper (CPU performance in particular) had a rather large variance; to counteract this uncertainty, we perform 10 iterations of each run and compute the average of the relevant results. In total, we performed 100-120 runs to gather data.

For the single-device runs, we ran MemXCT with 24 iterations on the two significant input datasets, CDS1 and

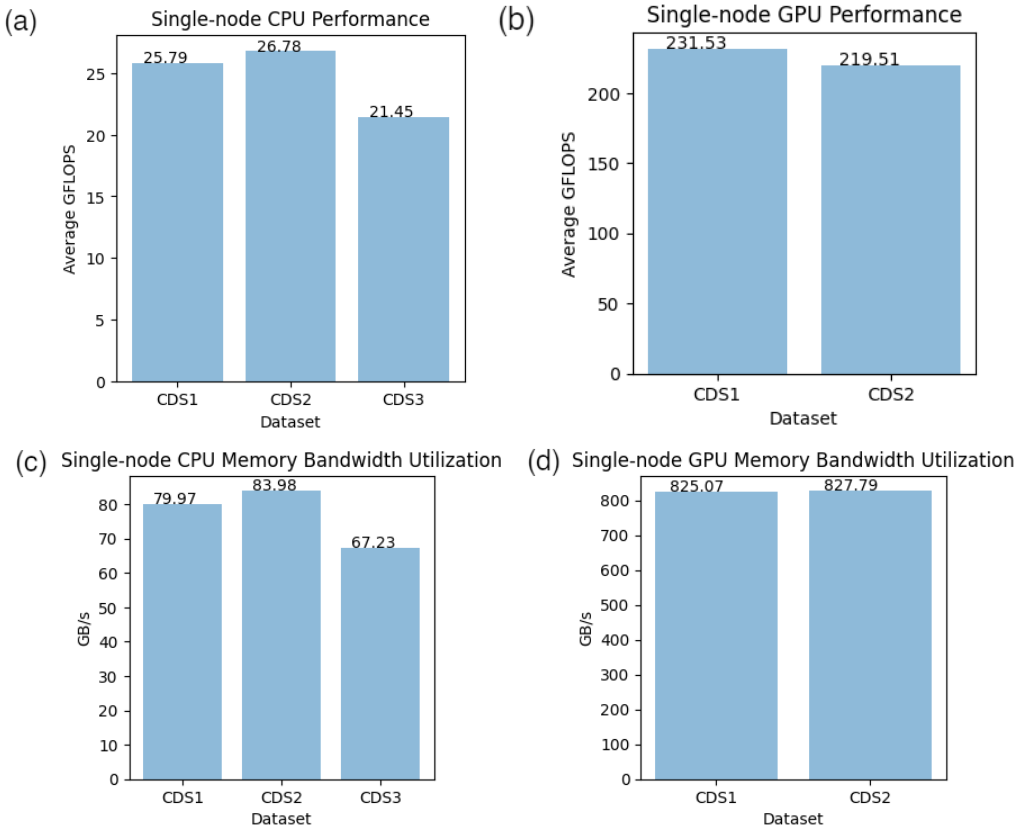

Fig. 1. Single-node performance comparison on CPUs and GPUs, including average GFLOPS and memory bandwidth.

CDS2. For the single-CPU runs, we used 60 OMP threads on a single socket, all within a single MPI rank. For the single-GPU runs, we used a single thread on the CPU. As the CPU only performs preprocessing in the GPU code, the number of threads allocated to single-GPU runs is not significant for our measurements. For each of these 40 runs, we considered the reported average GFLOPS performance as well as the total memory bandwidth utilization across projection and backprojection. These measurements are analogous to those from Figure 9 of the MemXCT paper.

For the multi-node scaling experiments, we ran MemXCT with 24 iterations on CDS1 and CDS2, on 1, 2, or 4 nodes. In the interest of budget and sharing SKU's (Stock Keeping Units, an Azure term for a compute resource), we did not perform 8-node scaling experiments. For each of these 60 runs, we considered the total reconstruction time and the total time of the partial projection (A_p), communication (C), and reduction (R) kernels. Since forward projection and back projection are closely related, for each of the three kernels, we measure the total time of the kernel in both forward- and back-projection.

For the CPU runs, we used the HB120rs_v2 SKUs. Each node has a large memory size, so runs of any of the CDS datasets can fit in a single node's memory. The high core count allows us to benefit from the speedup of small domain partitions while not incurring a communication bottleneck. These nodes have reasonable memory bandwidth at 350 GB/s, but still less than the 400 GB/s bandwidth on the KNL nodes used in the MemXCT paper. Therefore, we should still observe memory bandwidth to be a significant bottleneck. The nodes also have a 32KB L1 cache, so we expected to observe satisfactory results with similar tuning parameters to those given in the MemXCT paper. One notable shortcoming is the AMD nodes, which only support AVX2 [5]. As a result, automatic compiler vectorization results are disappointing. This has large performance implications, as we will discuss in the experimental run section.

For GPU runs, we used NC24rs_v3 SKUs. These nodes contain 4 Nvidia Tesla V100 GPUs, which we chose because the V100 consistently had the best performance out of the three GPUs tested in the MemXCT paper [3]. Since CPUs have very little impact on performance of the GPU runs, we should expect nearly identical single-GPU performance to the runs in the MemXCT paper. Note that in the interest of time, we only analyze the performance of CDS1 and CDS2 on GPUs, just as the paper only analyzes ADS1 and ADS2 on GPUs.

On the software side, compiling is straightforward, requiring a C++ compiler, an MPI library, and CUDA on GPU nodes. As discussed later, the automatic vectorization results are very poor in critical areas; it is possible that one could achieve better results by inserting manual vectorization pragmas, though we did not have time for this. Runs were scheduled using the Slurm scheduler, adjusting the number of nodes in each run. For CPU runs, we ran 60 OMP threads per MPI rank, with one rank per socket. GPU runs used one MPI rank and OMP thread per GPU.

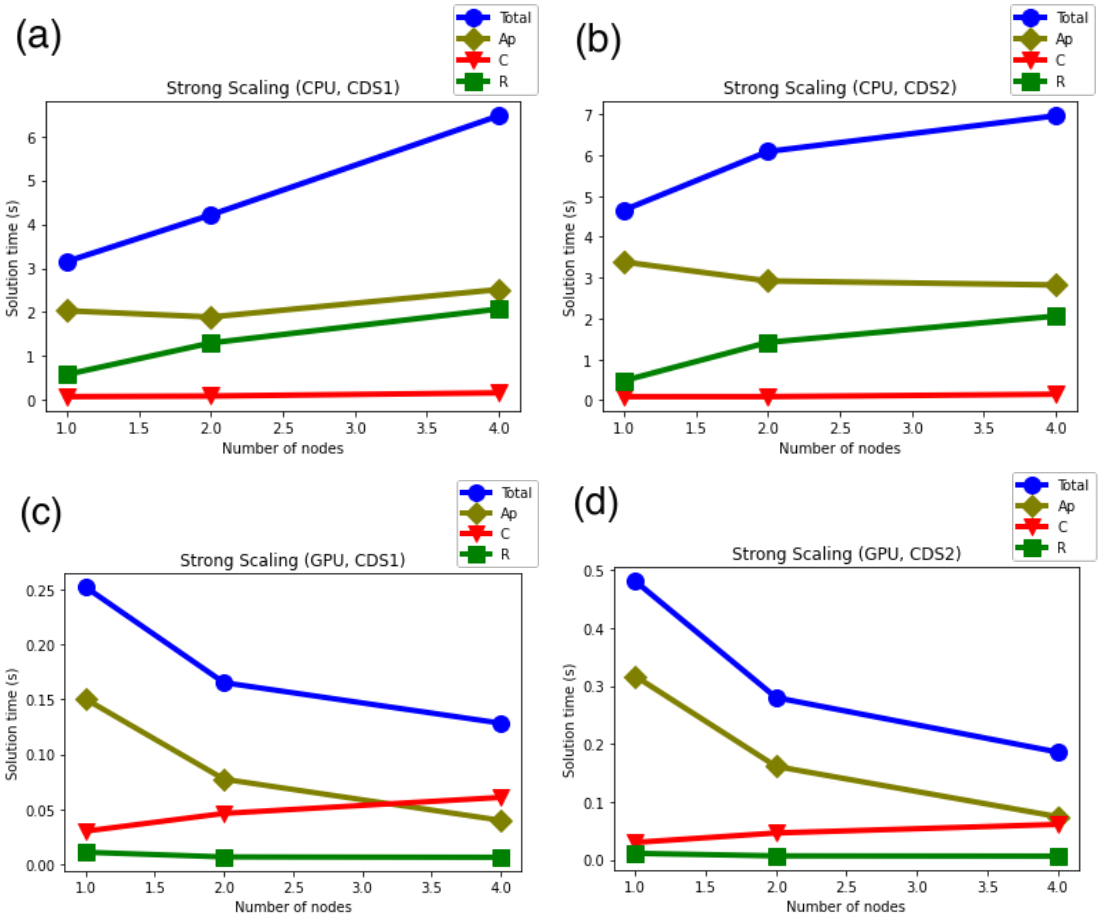

Fig. 2. Strong scaling of multi-node CPU and GPU runs on CDS1 and CDS2

4 Single CPU vs. Single GPU Performance Comparison

Figure 1 shows mixed results in reproducing the results from the MemXCT paper. The GPU performance is just above 200 GFLOPS on average, which mirrors that of the GPU performance in the MemXCT paper for datasets of the same size. The CPU performance for large datasets, namely CDS3, is also comparable to the performance from the paper. However, we were unable to reproduce the CPU performance for smaller datasets. With a slightly higher clock speed but a smaller vector size than that of KNL, we predicted ˜80 GFLOPS on a dataset the size of CDS1, but our runs were severely bottlenecked by multiple performance concerns.

One of the most unexpected performance differences came from the CPUs architecture. The HB120rs_v2's AMD processors only supports AVX2 vectorization, as opposed to Intel's more powerful AVX-512 [5]. Unlike AVX2, AVX-512 supports vectorized scatter-gather instructions [6], [7]. Given this restriction, GCC is unable to automatically vectorize performance-critical loops. In particular, the following loop is taken from the optimized kernel in Listing 3 of the MemXCT paper, which occurs twice in the source code (`kernels.cpp`, lines 82 and 160).

```
for(int i = displ[start+j]; i < displ[start+j+1]; ++i)
  output[j] += input[ind[i]]*val[j];
```

The inner loop above contains an indirect access: `input[ind[i]]`. Transforming a scatter operation to the gather operation above, combined with optimizations to increase cache performance, increases performance in the MemXCT kernel. Intel AVX-512 contains instructions that are able to vectorize this loop, and Intel KNL's vector processing unit in particular can perform this operation in a single intrinsic [7]. However, we cannot take advantage of this using GCC with AVX2 [7]. In fact, many loops in the codebase are not vectorizable using AVX2. This significantly decreases the performance gains observed in Figure 9 of the MemXCT paper. For example, the MemXCT paper sees significant performance improvement on the ADS2 dataset, which has a relatively low baseline performance. On the CDS1 dataset of the same size, we observe only approximately 30 GFLOPS on average, which is is only slightly above the baseline performance on KNL.

Other factors are memory latency and memory bandwidth. Although the reported maximum possible memory bandwidth on our SKU is 350 GB/s, we only achieved utilization of approximately 80 GB/s (22.9% utilization), with a maximum of 107 GB/s (30.6% utilization).

Despite this performance degradation, on CDS3, we observe performance comparable to the MemXCT paper. For larger datasets, optimization bottlenecks are largely overtaken by the high L2 miss rate and limited memory bandwidth, and the lack of vectorization is no longer a

significant factor. We observe a similar average GFLOPS and memory bandwidth utilization as that of ADS4 on KNL in the MemXCT paper.

Our GPU results closely mirror those of the MemXCT paper, given a very similar setup with the same model of GPU.

5 Strong Scaling on GPUs and CPUs

For the strong scaling study, we ran MemXCT with 24 iterations on 1, 2, and 4 nodes, on both datasets. Our setup was slightly different than the one used in the MemXCT paper, due to the SKU architectures: on the CPU version, every node used 2 MPI ranks with 60 OMP threads each, as the HB120rs_v2 nodes contain two sockets with 60 CPUs each [2]. For GPUs, we used 4 MPI ranks per node, one for each GPU. Every rank had a corresponding CPU thread, though CPU performance on GPU runs is inconsequential. A small modification was made to the GPU code to properly assign GPU devices in this setup.

Figure 2 shows the results of the strong scaling experiments, which are also mixed. A_p, C, and R refer to the three steps of forward- and back-projection: the partial forward projection, communication, and reduction operations respectively. CPU runs in particular showed surprisingly fast times on only one node, with slightly degrading performance on more nodes. The architecture of the HB120rs_v2 greatly affects the strong scaling studies. Due to the large memory size on an individual node, it's possible that the entire problem already fits in memory on a single node. This negatively affects scaling, as a large number of cores would not have enough individual work on small datasets. This would explain the speedup on smaller datasets and the reduced performance gain from adding more nodes. Furthermore, it may be possible that a large number of cores sharing a common cache would increase the rate of cache misses.

It is possible that this setup caused us to be slowed down by communication as we add more nodes: we observed that using more than one MPI rank on a single node starts to degrade the performance. These irregularities may disappear if the scaling experiments were continued to more nodes or if the scaling experiments were performed on larger datasets. This is reinforced by the fact that the performance degradation from 2 to 4 CPU nodes is less severe with CDS2 than with CDS1. In addition, we notice a much faster time for the communication kernel than expected, as the SKU uses high-speed InfiniBand and RDMA.

The GPU scaling proceeds largely as expected, and resemble Figure 11 from the MemXCT paper much more strongly. The speedup on several nodes scales somewhat slower than expected; this may be because the tested datasets are relatively small, and the speedup would become more apparent with larger datasets.

6 Conclusion

We found that the CPU MemXCT performance does not translate from Intel to AMD because of the lack of AVX-512. The availability of a vectorized scatter-gather instruction in KNL allows for the indirect memory accesses in the performance-critical loops to be much faster on Intel.

In our strong scaling CPU runs, we found higher relative performance on single node runs. We believe this is because the high core count of the HB120rs_v2 nodes is sufficient for smaller data sets. Additionally, adding more nodes introduces a communication bottleneck.

In strong scaling experiments on GPU nodes, we observe performance on par with the MemXCT paper due to similar hardware configuration used in our experiments.

Through this work, we confirmed that MemXCT delivers in offloading the bottleneck from compute to width as it was promised in the original paper. However, the performance gains may vary depending on the hardware used, specifically the presence of the AVX-512 instruction set.

7 Artifact Availability

https://github.com/nprindle/MemXCT is the public link to our artifact. It contains our collected data, run scripts, figure generation code and more.

References

[1] M. Hidayetoğlu, T. Biçer, S. G. de Gonzalo, B. Ren, D. Gürsoy, R. Kettimuthu, I. T. Foster, and W.-m. W. Hwu, "Memxct: Memory-centric x-ray ct reconstruction with massive parallelization," ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356220

[2] Microsoft, "Hbv2-series," https://docs.microsoft.com/en-us/azure/virtual-machines/hbv2-series, accessed: 2020-11-11.

[3] Microsoft, "Ncv3-series," https://docs.microsoft.com/en-us/azure/virtual-machines/ncv3-series, accessed: 2020-11-11.

[4] Microsoft, "Ddv4 and ddsv4-series," https://docs.microsoft.com/en-us/azure/virtual-machines/ddv4-ddsv4-series, accessed: 2020-11-11.

[5] AMD, "Amd epyc 7xx2-series processors compiler options quick reference guide," https://developer.amd.com/wordpress/media/2020/04/Compiler%20Options%20Quick%20Ref%20Guide%20for%20AMD%20EPYC%207xx2%20Series%20Processors.pdf, accessed: 2020-11-11.

[6] M. J. Buxton, "Haswell new instruction descriptions now available," https://software.intel.com/content/www/us/en/develop/blogs/haswell-new-instruction-descriptions-now-available.html, accessed: 2020-11-11.

[7] J. Reinders, "Intel avx-512 instructions," https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html, accessed: 2020-11-11.