Reproducing MemXCT during the 2020 Student Cluster Competition

Team Phoenix, Team 11[1]

Georgia Institute of Technology

Abstract

For the 2020 Student Cluster Competition, we are attempting to reproduce 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 used for 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 to clusters by moving much of the overhead to memory bandwidth and memory latency. MemXCT also applies several key optimizations in order to ease memory pressure. In this paper, we test the performance of MemXCT on single CPUs and GPUs, and run initial strong scaling experiments on multiple nodes of each. We confirm the impact of MemXCT's optimizations, and find that some optimizations do not translate well to the AMD processors of our CPU nodes. We also confirm the findings of the MemXCT paper on Tesla V100 GPUs. As we use GCC and OpenMPI, we test their performance against the original findings.

Keywords: Reproducible computation, Student Cluster Competition, MemXCT

[1]akazmi30@gatech.edu, aman.jain@gatech.edu, albertchen@gatech.edu, marissasorkin@gatech.edu, nprindle@gatech.edu, sudhanshu@gatech.edu, thakkarv@gatech.edu

1. Introduction

The algorithm introduced by the MemXCT [1] paper is notable in that it moves many bottlenecks to memory bandwidth and latency. 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. However, this leads to many indirect accesses. MemXCT therefore applies several optimizations in order to help cache performance and ease memory pressure from indirect accesses; it performs a pseudo-Hilbert ordering of the input domain for better cache locality, and it uses multi-stage buffering to pre-load indirect accesses into L1 cache. We plan to reproduce the results from the paper on both CPUs and GPUs. As we are using AMD CPUs with poor vectorization capabilities, 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 hope to confirm that MemXCT scales easily to larger clusters as the problem size per device decreases.

2. Experimental Setup

The master node of our CPU cluster was Azure's D12_v2 node. The execute nodes were Azure's HB120rs_v2 nodes, which are dual socket nodes containing AMD EPYC 7V12 64-core processors. Each node has 480 GB of memory, with a theoretical bandwidth of 350 GB/s. The master node of our GPU cluster was Azure's D64ds_v4 node, for compatibility with the execute nodes. The execute nodes were Azure's NC24rs_v3 nodes, each with 24 cores and 4 NVIDIA Tesla V100 GPUs.

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, as our compiler 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.

The datasets used are the three provided competition datasets: CDS1, CDS2, and CDS3.

Other than libraries, there were minimal modifications made. The most major modification was to the GPU code; by default, the code did not assign device numbers properly when run across multiple nodes with multiple GPUs each.

3. Description of experimental run

For the experiment, we implemented two 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. Note that 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 planned to perform 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 CDS2. For the single-CPU runs, we used 60 OMP threads on a single socket, all under a single MPI rank. For the single-GPU runs, we used a single thread that utilized a single GPU. 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 either 1, 2, or 4 nodes. In the interest of budget and sharing SKUs, we did not perform 8-node scaling experiments. For each of these 60 runs, we considered the total reconstruction time, as well as 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 high core count and large memory size, so runs of any of the datasets can fit in a single node's memory. The high core count hypothetically allows us to benefit from the speedup of small domain partitions, while not incurring a communication bottleneck. Compared to other SKUs, its memory bandwidth is fairly good at 350 GB/s, but as it is still lower than the 400 GB/s bandwidth on the KNL nodes used in the MemXCT paper we should still observe memory bandwidth to be a significant bottleneck. The nodes also have a 32KB L1 cache, so we should observe good results with similar tuning parameters to those given in the MemXCT paper. One notable shortcoming is the AMD nodes, which only support AVX2. As a result, automatic compiler vectorization results are disappointing. This has large performance implications, as we will later discuss.

For our GPU runs, we used NC24rs_v3 SKUs. These nodes contain 4 Nvidia Tesla V100 GPUs, since the V100 consistently had the best performance out of the three GPUs tested in the MemXCT paper. 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 very simple, only requiring a C++ compiler, an MPI library, and on GPU nodes, CUDA. 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 for 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.

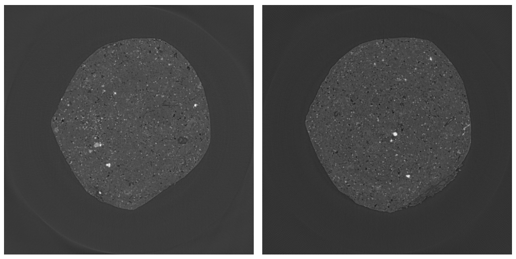
Figure 1: The tomograms for CDS1 (left) and CDS2 (right) as reconstructed by MemXCT

4. Single CPU vs. Single GPU Performance Comparison

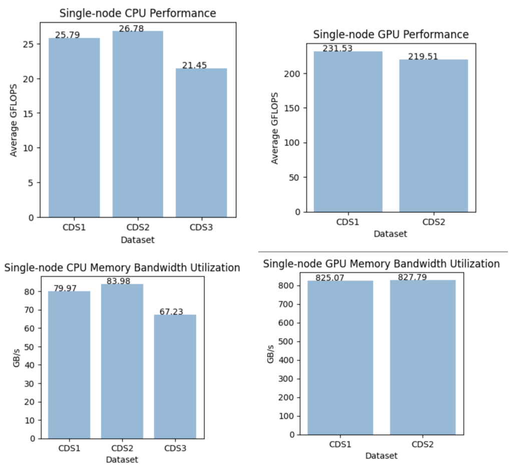

Figure 2: Single-node performance comparison on CPUs and GPUs, including average GFLOPS and memory bandwidth.

In terms of reproducing the results from the MemXCT paper, Figure 2 shows mixed results. 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 AVX512. Without explicit pragmas, 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 AVX512 contains instructions that are able to vectorize this loop, but this is not possible using GCC with AVX2. 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.

Another concern is memory latency and memory bandwidth. Although the reported maximum possible memory bandwidth on this SKU is 350 GB/s, we only achieved utilization of approximately 80 GB/s, with a maximum of 107 GB/s. We observed a significant drop in CPU utilization on every iteration, likely due to being bottlenecked by memory accesses.

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

As expected, the 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

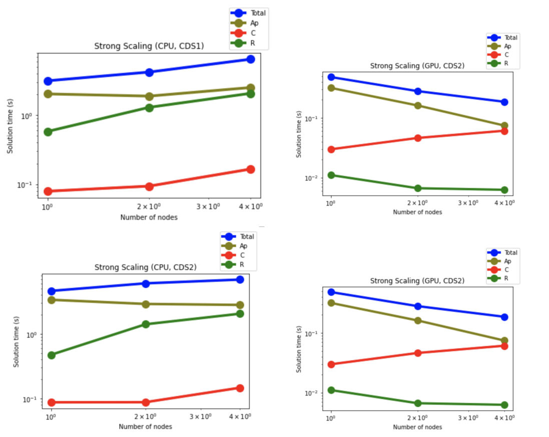

Figure 3: Strong scaling of multi-node CPU and GPU runs on CDS1 and CDS2

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. 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. Note that a small modification was made to the GPU code to properly assign GPU devices in this setup.

Figure 3 shows the results of the strong scaling experiments, which are also mixed. 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 high core count it's possible that the entire problem already fits in memory on a single node. This would explain the speedup on smaller datasets and the reduced performance gain from adding more nodes.

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 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 the higher relative performance on single node runs. We believe this may be because the high core count of the HB120rs_v2 nodes might be more than enough for smaller datasets, and adding more nodes introduces an MPI bottleneck.

With GPU strong scaling, the performance matches our expectations because our hardware is similar to the hardware used in the MemXCT paper.

Through this process, we were able to confirm that offloading the bottleneck from compute to memory bandwidth does make MemXCT perform better than traditional XCT algorithms. However, the performance gains are largely dependant on the hardware used, specifically the presence of the AVX-512 instruction set.

References

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