Distributed GPU Acceleration for miniFE

Ryan Greenblatt

May 2019

1 Background

miniFE provides an example application for distributed and threaded CPU finite element simulations[1]. Outside of data setup, miniFE uses the conjugate gradient method for solving a sparse linear system. While the data setup and initialization could be further parallelized and some parts could be effectively moved to the GPU, focus was placed on porting the conjugate gradient method to run distributed on GPUs. The algorithm for the method is detailed below. The kernels involved in applying the method to a sparse system include the dot product, sparse matrix dense vector multiplication (SpMV), and WAXPBY (element wise multiplication). As such, this algorithm is well suited to GPU acceleration, but is certainly going to be memory bandwidth limited.

```
procedure CONJUGATE GRADIENT METHOD($A, x, b, \epsilon$)
    $r_0 \leftarrow b - Ax$
    $r_1 \leftarrow r_0$
    $p \leftarrow r_0$
    $i \leftarrow 1$
    while $\|r_i\|_2 > \epsilon$ and $i \leq$ max_iter do
        $p \leftarrow r_i + \frac{\|r_i\|_2^2}{\|r_{i-1}\|_2^2} p$
        $\alpha \leftarrow \frac{\|r_i\|_2^2}{\langle Ap, p \rangle}$
        $x \leftarrow x + \alpha p$
        $r_{i+1} \leftarrow r_i + \alpha Ap$
        $i \leftarrow i + 1$
```

2 Approach

While the ELLPACK data format is hypothetically implemented in miniFE, the original code failed to compile when using that format. As such, the CRS format was used exclusively. OpenMP is used extensively for CPU threading in the original code. Because of this, it would have been desirable to use the clang compiler's support for generating PTX instructions from OpenMP directives. However, support is relatively new and a known linker

[1]The most recent version on Github has CUDA support, but the variant I worked with had no CUDA support

bug with no documented solutions made this challenging, so this approach was not used.

The cuSPARSE and cuBlas libraries were used for dot products and SpMV as they have highly optimized implementations for these simple operations.

The existing code makes heavy use of STL vectors, so a custom allocator was used to allocate the vector array as CUDA managed memory. Managed memory is advantageous for improving interactions with MPI and making incremental implementation easier.

Due to a lack of access to systems with GPU interconnect[2], I used generic MPI message passing through the CPU.

I have modified miniFE so that the original types are no longer configurable, only int ordinals are used. This was done because a fixed ordinal type is needed for cuSPARSE and cuBlas. The scalar type is still configurable and I tested single and double precision.

Code is attached in the email. I have included a CMakeLists.txt file which can be used to compile the code on the CCV. The `cuda/10.0.130`module and some MPI module must be loaded. Note that gencode is specified to be “arch=compute_70,code=sm_70” which may cause issues on older GPUs.

The original miniFE code had a bug that would cause occasional segmentation faults when running multiple processes and several incomplete and untested features which weren't clearly labeled as such. In general, the code felt unfinished. I have tried to polish things somewhat and remove partially implemented features.

[2] I didn't want to spend a lot of money on cloud computing

3 Results

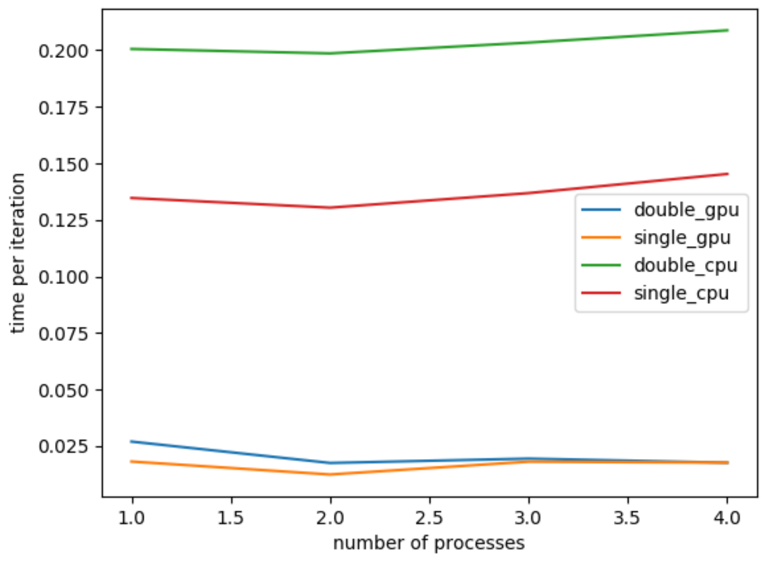

Figure 1: GPU and CPU iteration timings for various numbers of processes and different data types. Note that the number of process also corresponds to the number of GPUs used for the GPU computations.

Because of the 2 GPU limit I have on the CCV, testing was conducted on a system with 4 2080 Tis (using a peer to peer cloud provider). CPU only testing was done with 16 threads. Access to more GPUs would have been ideal to better understand scaling, but options are limited without substantial spending. Note that the setup and initialization take the vast majority of the time when using a GPU on any problem size which I could feasibly test. This time isn't included in the results. I tested with all dimensions equal (a cube) and I tested single and double precision. Single and double precision appeared to have similar residuals in this case.

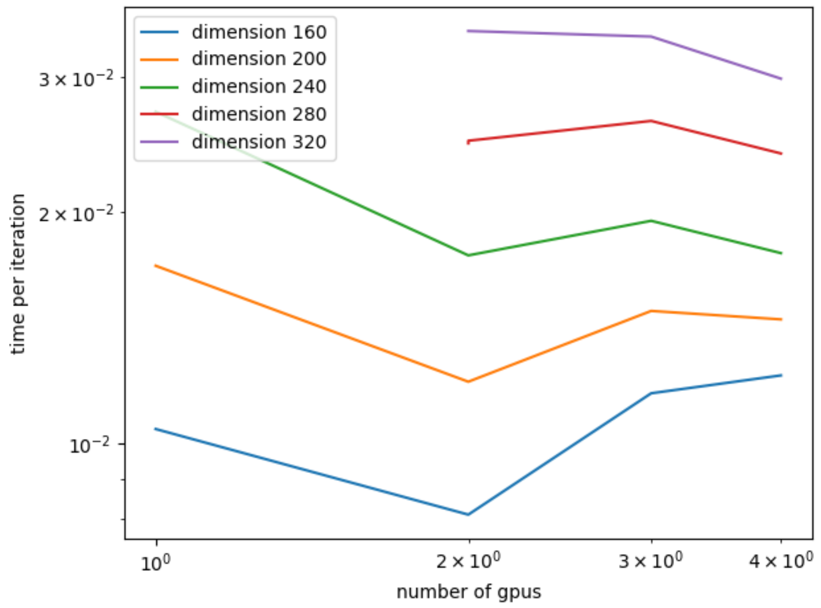

Figure 2: Log-log strong scaling plot for various dimensions with double precision. Not all dimensions could be successfully run with 1 GPU.

The GPU implementation is almost an order of magnitude faster than the CPU implementation. Adding an additional GPU scales the speed of the computation by almost 2, but further additional GPUs actually resulted in small reductions in speed. However, memory is likely to often be limited, so more than 2 GPUs may be needed for larger problem sizes. Because managed memory is used, when memory runs out on the GPU(s), the computation continues but is greatly slowed.

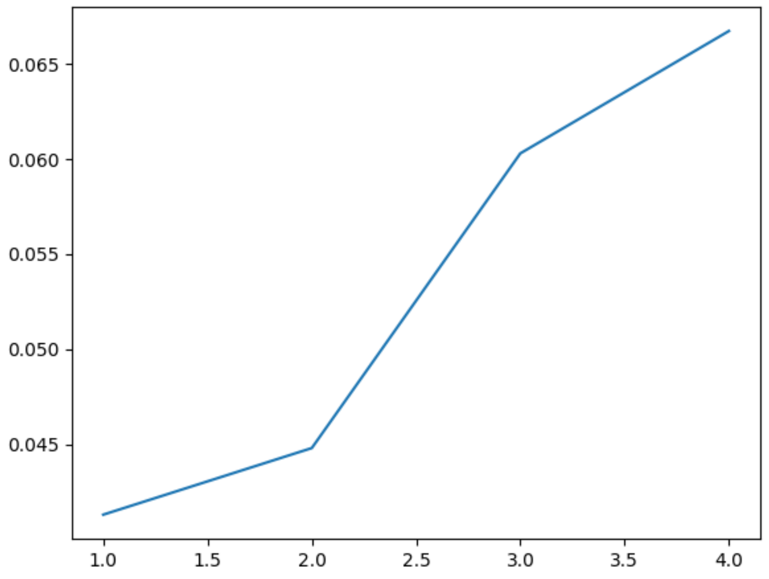

Figure 3: Weak scaling plot. A increase in the dimension of the problem results in approximately a cubic increase in number of FLOPS, so the following dimensions were used for 1, 2, 3, and 4 GPUs respectively: 280, 353, 404, and 445.

When evaluating weak scaling, I only scaled the quantity of GPUs. The total number of CPU threads remained constant.

4 Conclusions

Communication which has to go through the CPU quickly became the limiting factor when using multiple GPUs. I would imagine that with effective use of GPU interconnect, the scaling would be far better. It would likely be possible to optimize the CUDA kernels by actually fusing the dot product and SpMV operations in the fused kernel rather than executing them sequentially. Another potential optimization would be to try half precision.