Max Gardos
AM 148

Optimizing Linear Algebra on GPUs: A Comparative Study with HIP C++

Background

Linear algebra routines are foundational algorithms on which modern scientific computing runs. On large datasets, they can be very computationally intensive. Parallel routines provided by the HIP C++ API Runtime and Kernel for GPUs can greatly increase the speed at which modern computer clusters can compute these routines. Additionally, prewriting optimized C++ functions to perform these commonly used routines saves time and effort when implementing them in a new project.

Linear Algebra Routines

The linear algebra routines I will implemented are as follows. I was originally going to create a parallel LU decomposition routine and a triangular matrix solver, but I was not able to come up with a way to effectively parallelize any of the serial algorithms I encountered in my research.

`float dasum(const double* x, double* y, unsigned int n)`

This function performs the computation

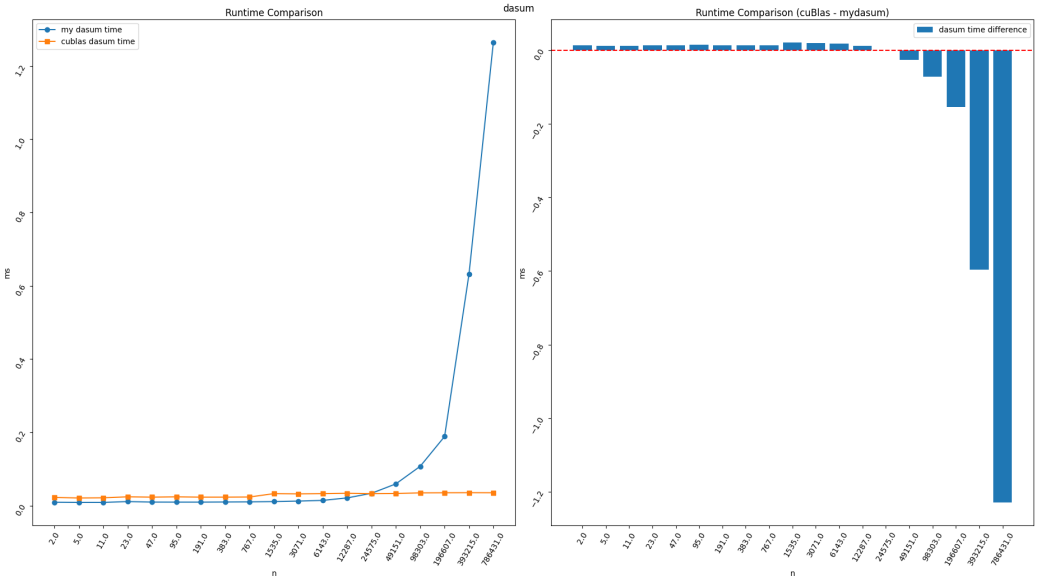
$$y = \sum_{i=0}^{i=n} |x[n]| \tag{1}$$

which saves the absolute sum of the vector `x` in `y` and returns the total time it took for the reduction kernel to run in ms.

Essentially, each thread loads an element from global memory into a shared memory buffer, and a sum reduce is performed on that memory buffer. After all threads are synced at the end, the reduced sum is written out to `tid` 0.

This routine, and the following `dnrm2` routine, utilize an optimized parallel reduction method outlined by Mark Harris in his CUDA webinar *Optimizing Parallel Reduction in CUDA*. For an array of size n, the step complexity of "halving" the array in shared memory is $\mathcal{O}(log(n))$, and the complexity of adding every element to each other in the array is $\mathcal{O}(n)$. For p threads, this means that this algorithm runs in $\mathcal{O}(\frac{n}{p} + log(n))$.

To test this function, I compared it against its cuBlas equivalent, `cublasDasum()`. For both functions, I averaged how fast it took them to sum an input vector of ones in the range $n = 2 -> 2^{20}$ over 9 iterations. I doubled n and added one to it each step to make sure that the routine worked with sizes that the block size and grid sizes of of 128 and `(n + dimBlock.x - 1) / dimBlock.x` would not tile neatly into. My implementation is faster until $n = 12287$, at which point it very quickly becomes more inefficient. This pattern occurs with every other routine except `dcopy()` - I suspect that this is due to the way that the input array is completely loaded into device memory with one call to `hipMalloc()`, loading blocks of it and performing the reduce on those before summing every block would likely be faster.



`float dnrm2(const double* x, double* y, unsigned int n)`

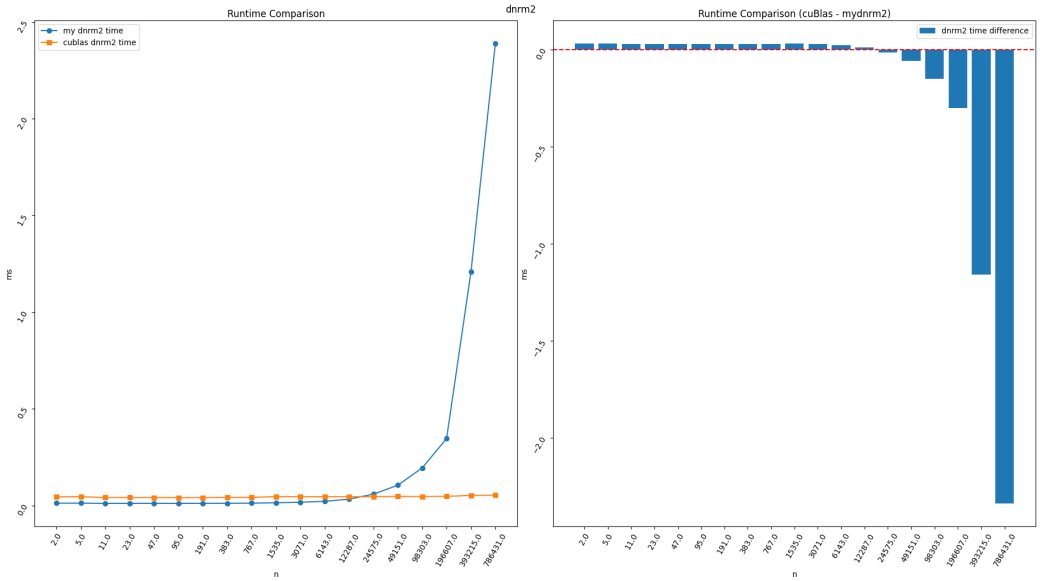
This function performs the computation

$$y = \sqrt{\sum_{i=0}^{i=n} |x[n]|^2} \tag{2}$$

which saves the square root of the absolute sum of the squared elements of vector `x` in `y` and returns the total time it took for the reduction kernel to run in ms.

First, a map taking $\mathcal{O}(\frac{N}{P})$ time transforms $x[n]$ into $|x[n]|^2$ by taking the absolute value and then squaring each element of x. Then, the sum reduction kernel over the transformed x has the same runtime complexity as `dasum()`. At the end, the square root is taken of the resultant sum, taking $\mathcal{O}(\infty)$ time. So, in total this routine has time complexity $\mathcal{O}(\frac{N}{P}) + \mathcal{O}(\frac{n}{p} + log(n)) + \mathcal{O}(1)$.

To test this function, I compared it against its cuBlas equivalent, `cublasDnrm2()`. For both functions, I averaged how fast it took them to take the euclidian norm of an input vector of ones in the range $n = 2 - > 2^{20}$ over 9 iterations. This implmentation has a constant block size of `64` and grid size of `128`. Like with `dasum()`, my implementation is faster until $n = 12287$, at which point it very quickly becomes more inefficient. One issue that I encountered with both this and the previous routine was the sum reduce kernel would overflow for large matrices, which I think could be fixed if the routine was performed on tiles of the input array or each element of $|x[n]|^2$ was multiplied by an approximation constant instead of performing a single square root at the end.



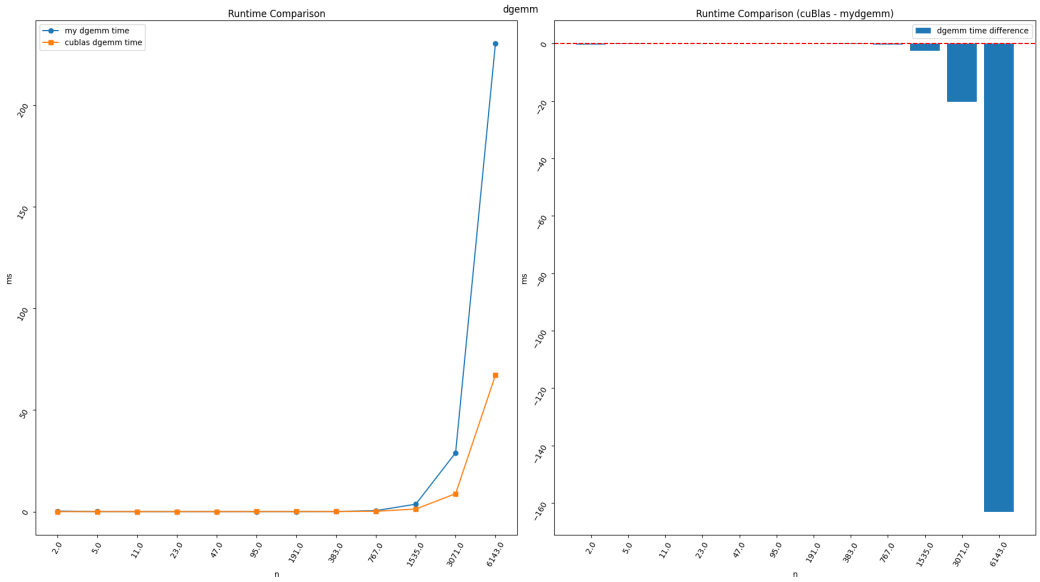

```
  float dgemm(const unsigned int m,
const unsigned int n,
const unsigned int k,
const double alpha,
double* A,
double* B,
const double beta,
double* C)
```

This function, a general matrix multiplication, performs the computation $C = a*AB+b*C$, where A is an `m x k` matrix, B a `k x n` matrix, and C a `m x n` matrix. My implementation is based on the matrix multiplication kernel described in the shared-memory chapter of the *CUDA Toolkit Programming Guide*.

Essentially, matrices A and B are first divided evenly into tiles of size 32 * 32, and each thread block, after loading one tile from A and one tile from B into shared memory, computes the matrix product between those tiles and saves it in C.

One issue I ran into was finding matrices whose dimensions did not evenly divide the block size of 32. In order to remedy this, I padded the dimensions of matrices A, B, and C with zeros so that m, n, and k all were resized to the smallest multiple of 32 larger than each of them.

To test this function, I compared it against its cuBlas equivalent, `cublasDgemm()`. For both functions, with $\alpha = 1$, $\beta = 0$, and A as an identity matrix, I averaged how fast to perform the matrix product on square `n x n` matrices A, B, and C for n in the range $n = 2 -> 8192$ over 9 iterations. To assert that $B = C$ after $A * B = C$, I took the sum of B and asserted that it was equal to the sum of C after the calculation. I was not able to perform the product on matrices larger than this due to issues with running out of memory on Lux - I suspect a strided memory approach would not have this problem. My implementation was only faster for very small matrices, and was much slower for the largest ones.



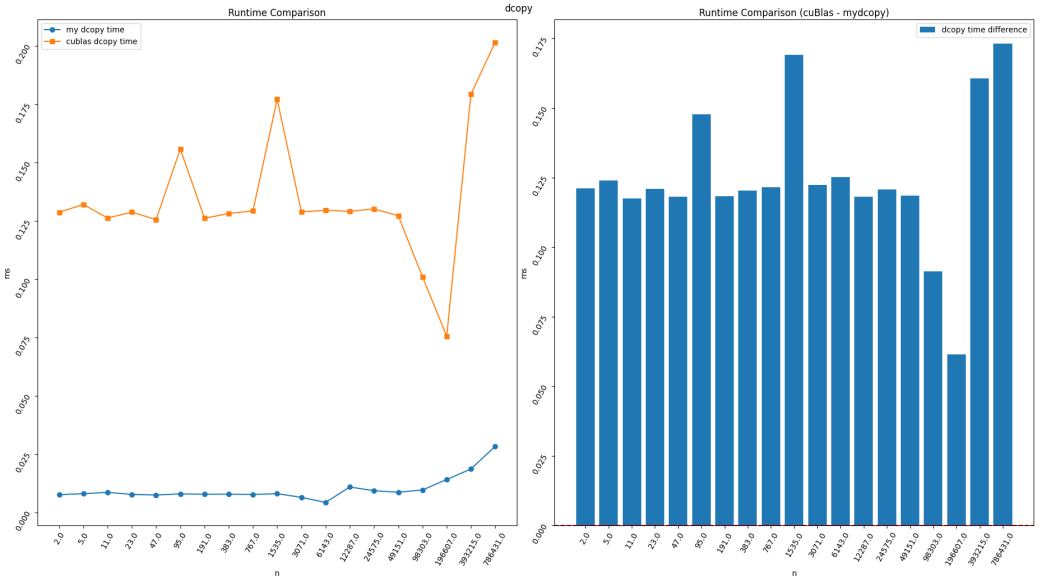
`float dcopy(int n, double* x, double* y)`

This function copies an array x with n elements into the array y which can hold n elements. My implementation of this routine and the following `daxpy()` routine uses a grid stride loop taken from the article *CUDA Pro*

Tip: Write Flexible Kernels with Grid-Stride Loops written by Mark Harris for the NVIDIA Developer Technical Blog.

This copy is a simple map which loops over the grid stride of the thread block to copy each element $x[i]$ into $y[i]$. I used a small block size of 64 and grid dim `(n + blockSize + 1) / blockSize`. With p threads and n elements, this copy algorithm should take $\mathcal{O}(\frac{n}{p})$. I experimented with a tiled shared memory implementation like that described in *An Efficient Matrix Transpose in CUDA C/C++* written by Mark Harris for the NVIDIA Developer Technical Blog but struggled to implement padding of the arrays like I did for `dgemm()` and ultimately chose to go for the simpler approach.

To test this function, I compared it against its cuBlas equivalent, `cublasDcopy()`. For both functions I averaged how fast it took to copy an array of ones x to an array y, both of length n, for n in the range $n = 2 -> 8192$ over 9 iterations. I took the sum of y each time to assert that it was equal to n. Surprisingly, this function was faster than `cublasDcopy()`, which I suspect is due either an error on my part or an accidental extreme utilization of memory coalescence which the grid strided loop helps create.

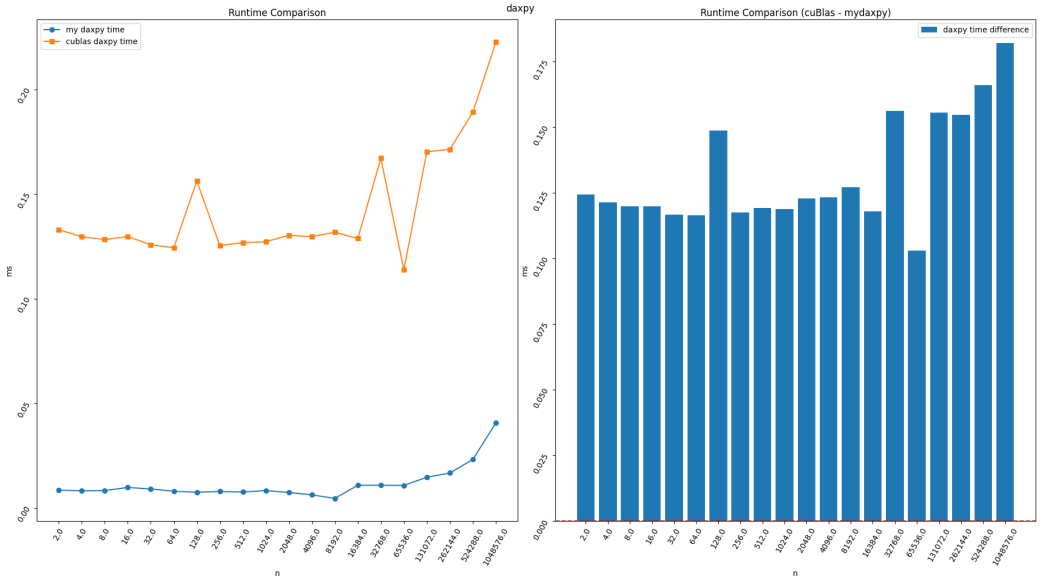


`float daxpy(int n, double alpha, double* x, double* y)`

This function performs the computation $y = a * x + y$, where y and x are vectors of length n. This routine also uses a grid-stride loop.

This copy is also a simple map which loops over the grid stride of the thread block to copy each element $a * x[i] + y[i]$ into $y[i]$. I used the same block and grid sizes that I did with `dcopy()`. With p threads and n elements, this copy algorithm should take $\mathcal{O}(\frac{n}{p})$.

To test this function, I compared it against its cuBlas equivalent, `cublasDaxpy()`. For both functions I averaged how fast it took to perform the calculation where $a = 2$, x is a vector of ones, and y is a vector of -1, where x and y both have length n, for n in the range $n = 2 - > 2^20$ with a step of $n \leftarrow n * 2 + 1$ over 9 iterations. I took the sum of y each time (as the result of the daxpy operation should result in a vector of ones with size n) to assert that it was equal to n. Similarly, this routine is faster than the cuBlas routine, likely for similar reasons that the `dcopy()` function is.



How to run the code

To run the code, make the `test` executable with the

```
make test
```
(3)

command, and then run

```
sbatch slurm.sh
```
(4)

to add the test executable to Lux's slurm queue. If you aren't using Lux, then you can run the file on a local HIP runtime with cuBlas.

Afterwards, run the python script `plot.py` with the command

```
python plot.py
```
(5)

to produce graphs of the generated `.dat` files.