ME759 Final Project

Diagonally Dominant Banded Systems Solving with CUDA

Elliott Biondo

December 15, 2013

Abstract

Partial differential equations are often solved numerically, which may involve a step where it is necessary to solve a system of equations that can be represented in the form

$$Ax = b,$$

where A is a diagonally dominant banded matrix and b provides multiple right-hand sides (RHS). In this project, a program was written to solve this system on an Nvidia GPU using CUDA. The program uses parallel implementations of LU decomposition, followed by forward and backward substitution. The program only stores and operates on the band of the matrix. Scaling analysis indicates that this program is systematically faster than the Intel MKL banded matrix solver (`dgbsv`) over a range of matrix dimensions and bandwidths and a single RHS. However, MKL outperforms the CUDA program when a large number of RHS are used. Visual profiling using NVVP indicates a potential for further parallelization of the forward and backward substitution steps, which dominant the execution time.

This report also explores an OpenMP banded solver program, which also only stores and operates upon the matrix band. It is shown that naively using OpenMP threads for the same purpose of CUDA threads does not result in speedup. The only meaningful parallelization achieved with OpenMP involved parallelizing the solving of multiple RHS. The resulting program is faster than the CUDA program for multiple RHS, but still slower than MKL.

Keywords: CUDA, linear system, band matrix, MKL, OpenMP

Contents

1 Introduction **3**

2 Algorithms **4**
2.1 Band Storage 5

3 CUDA Implementation **5**
3.1 Computational Methods 5
3.2 Usage 6

4 CUDA Numerical Experiments **7**
4.1 Scaling analysis and comparison with INTEL MKL 7
4.1.1 Intel MKL solver 7
4.1.2 Matrix dimension and bandwidth scaling comparison 7
4.1.3 Number of right-hand sides scaling comparison 8
4.2 Profiling with NVVP 8
4.3 CUDA memcheck 9

5 OpenMP **9**

6 Conclusion **10**

7 Appendix A: CUDA full memcheck results **12**

1 Introduction

Science, engineering, and financial applications often give rise to partial differential equations that are too difficult to solve analytically. This necessitates the use of numerical methods in order to obtain an approximate solution. Many such methods involve discretizing one or more variables in order to obtain a system of linear algebraic equations in the form

$$Ax = b, \tag{1}$$

where A is an $N \times N$ matrix, x is $N \times M$ matrix, and b is an $N \times M$ matrix. One way of solving Equation 1 is by LU decomposition. A is first decomposed into a lower triangular matrix (L) and an upper triangular matrix (U) such that

$$A = LU. \tag{2}$$

The system is then solved in two consecutive steps, first a forward substitution step

$$Ly_i = b_i, \tag{3}$$

then a backward substitution step

$$Ux_i = y_i. \tag{4}$$

In Equation 3, b_i is the ith column of b, which yields the solution to the intermediate vector y_i. In other words, Equations 3 and 4 must be solved M times to fully solve for x.

It is common for numerical methods to involve a constant A matrix and b vectors that changes over time (or some other variable). One such example would be unsteady state diffusion. In these cases, LU decomposition is advantageous because Equation 2 can be solved once, and then each time step will involve the less expensive solving of Equations 3 and 4. It is also common for A to be a banded matrix of bandwidth k. For the purpose of this report, k is assumed to be equal to

$$k = 2 \cdot k_{1/2} + 1, \tag{5}$$

where $k_{1/2}$ represents the upper/lower bandwidth, which are presumed to be equal. Banded matrices may arise if the operator that is being approximated by A contains a derivative term which is treated with some finite difference stencil. These systems are often strictly diagonally dominant, in other words:

$$|a_{ii}| > \sum_{i \neq j} |a_{ij}| \quad \text{for all i} \tag{6}$$

This project implements a parallel solution to Equation 1, presuming A is a diagonally dominant banded matrix and b has multiple right-hand sides (RHS). This is done using the aforementioned method on an Nvidia Graphics Processing Unit (CPU) using CUDA, and also on the CPU with OpenMP.

2 Algorithms

To tackle this problem, a set of mathematical algorithms was first chosen based on its potential to be parallelized [1]. For the chosen set of algorithms, the matrix is assumed to be strictly diagonally dominate. This eliminates the need for any pivoting operations. The pseudocode for the LU decomposition step is seen in Figure 1. This algorithm modifies the A matrix in place, such that the resulting L and U matrices are embedded within the original A. This avoids the burden of storing the zeros of the upper portion of L and lower portion of U.

Figure 1: Pseudocode for serial LU decomposition. Indexing is done from 0.

```
1  for i = 0 .. n-2
2    for j = i+1 .. n-1
3      A[j,i] = A[j,i]/A[i,i]
4    end
5    for j = i+1 .. n-1
6      for k = i+1 .. n-1
7        A[j,k] = A[j,k] - A[j,i]*A[i,k]
8      end
9    end
10 end
```

The forward solve for y continues as shown in Figure 2. The backward solve for x proceeds as shown in Figure 3. Both of these algorithms operate on the A matrix which is actually the superposition of the L and U, as created by the code in Figure 1. These algorithms solve the system in-place, meaning the the final answer for x is stored in b.

Figure 2: Pseudocode for the serial forward solve of Equation 3.

```
1  for i = 0 .. n-1
2    for j = i+1 .. n-1
3      b[j] = b[j] - b[i]*A[j,i]
4    end
5  end
```

Figure 3: Pseudocode for the serial backward solve of Equation 4.

```
1  for i = n-1 .. 0
2    b[i] = b[i]/A[i][i]
3    for j = 0 .. i-1
4      b[j] = b[j] - b[i]*A[j,i]
5    end
6  end
```

2.1 Band Storage

The algorithm described in Figure 1 preserves the band of A in creation of the L and U matrices. This allows this algorithm to utilize band storage. Using band storage, the algorithm must be implemented in a way such that operations are only done on the band. This complicates the indexing of the algorithms in Figures 1, 2, 3. By using band storage the number of stored entries is reduced from $N \cdot N$ to $N \cdot k$, as shown Figure 4.

Figure 4: Normal matrix storage vs. band storage.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & a_{00} & a_{01} & a_{02} \\ 0 & a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{53} & a_{54} & a_{55} & a_{56} & 0 \\ a_{64} & a_{65} & a_{66} & 0 & 0 \end{pmatrix}$$

3 CUDA Implementation

3.1 Computational Methods

The CUDA implementation utilizes the algorithm described in Figures 1, 2, 3, but only operating on the band of the matrix, which is stored in the band form described in Section 2.1. This presents numerous challenges to being parallelized with CUDA. For the LU decomposition step shown in Figure 1, iterative kernel calls were used for the outermost `for` loop. This was done in order to impose synchronization between iterations, which is necessary for this algorithm. This cannot be done using blocks because there is no notion of block synchronization in CUDA.

For each of these outer iterations, a $k_{1/2}$ by $k_{1/2}$ square chunk of the matrix is operated on as seen in Figure 5.

Figure 5: Progression of active chunk (red to blue).

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{pmatrix}$$

A 2D tiling strategy was used to perform these operations. This means that the largest number of operations that can be performed in parallel is $k_{1/2}^2$ which may be poor for small bands. For each outer iteration, 2 sequential steps must be performed:

1. Divide the first number in each row of the active chunk by upper-left-most element.

2. Recompute every other value in the active chunk (which depends on step 1).

Again, since there is no block synchronization in CUDA, two separate kernel calls were needed for steps 1 and 2. The step 1 kernel involves $k_{1/2}$ active threads. One thread in each block first retrieves the upper-left-most diagonal element from global memory and stores it in shared memory for the benefit of the block. Synchronization is performed and then every active thread must retrieve values from global memory from a different row in the matrix, which is not coalesced due to row-major storage. The step 2 kernel operates on the full $k_{1/2}$ by $k_{1/2}$ chunk. Memory access is coalesced (due to band storage) and no synchronization is required.

For each RHS the forward and backwards substitution steps must also be performed sequentially, which is done with two similar kernels. Each is called N times iteratively for each RHS. Upon each iteration, one thread per kernel call computes a final value and stores a multiplication factor in shared memory. Synchronization is done, then $k_{1/2}$ operations are performed in parallel, though memory access is not coalesced. Asynchronous memory copying is used for each RHS. All memory copy operations between device and host for the entire program utilize pinned host memory.

3.2 Usage

Full build and usage instructions are found in the file README_BUILD_AND_USAGE. The executable created is called `cuda_solve` and has two modes: one for testing and one for production runs. For testing, 3 arguments are supplied:

```
>> ./cuda_solve <matrix dimension> <bandwidth> <number of RHS>
```

This generates a random diagonally dominant matrix with the supplied dimension and bandwidth, and also the corresponding RHS (such that all values of x are unity), and solves the system of equations. The correctness of the solution is accessed and reported to `stdout`. Inclusive execution time is output. For production runs, 4 arguments are supplied:

```
>> ./cuda_solve <file name A> <file name b> <matrix dimension> <number of RHS>
```

The file containing matrix A must specify the values of A in row major order using band storage, whereas the file containing b must be in column major order. Timing results and the solution (x) are printed to `stdout`.

4 CUDA Numerical Experiments

4.1 Scaling analysis and comparison with INTEL MKL

All scaling analysis was done on the Euler cluster with a Nvidia GeForce GTX 480 GPU.

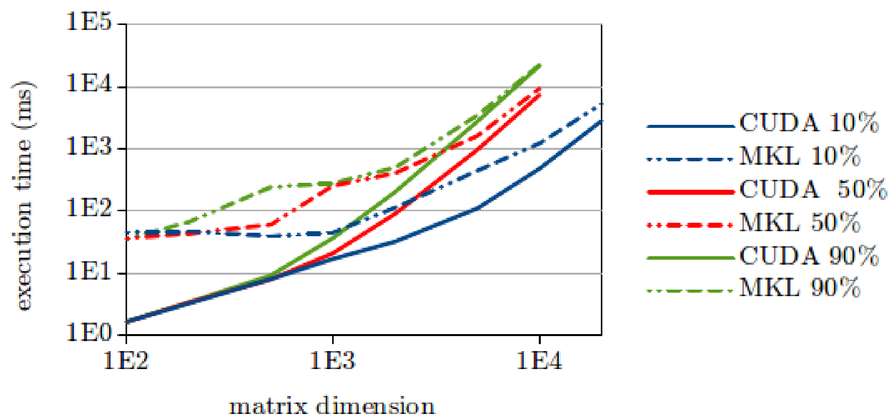
4.1.1 Intel MKL solver

A program was created using Intel MKL with identical functionality to the test mode of `cuda_solve` as described in Section 3.2. Full build instructions for the executable `mkl_solve` are found in the file README_BUILD_AND_USAGE. This script utilizes the MKL banded matrix solving kernel, `LAPACKE_dgbsv`.

4.1.2 Matrix dimension and bandwidth scaling comparison

Figure 6 shows how execution time scales as a function of matrix dimension (N) and bandwidth for a single RHS. The different bandwidths were used as a percentage of matrix dimension. This figure shows that the CUDA program is systematically faster than MKL over these dimensions and bandwidths. The plot suggests that MKL has a higher startup cost than CUDA, which has a diminishing affect on execution time for high matrix dimensions. The plot also shows that the discrepancy between CUDA and MKL is largest for small bandwidths, which is strange since the parallelization of the CUDA algorithm goes like $k_{1/2}$, as discussed in Section 3.

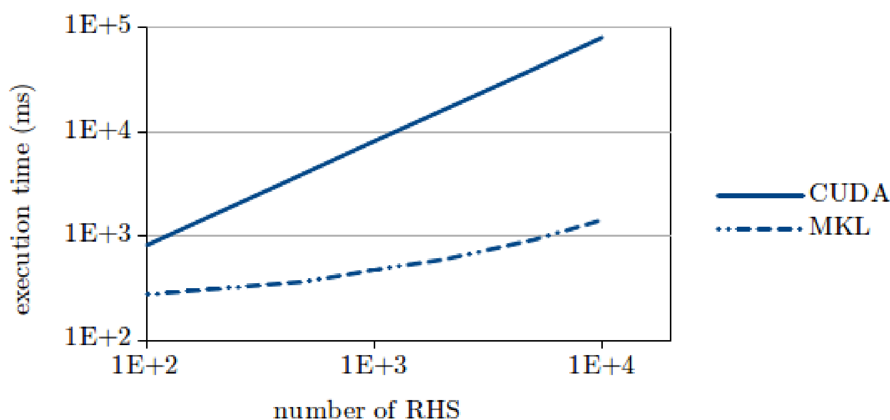
Figure 6: CUDA and MKL performance as a function of matrix dimension for different bandwidth percentages and a single RHS. Bandwidth percentages are approximate because bandwidths are odd (as defined by Equation 5). For example, for the N=100 cases, bandwidth of 11 was used for the 10% cases.



4.1.3 Number of right-hand sides scaling comparison

Scaling behavior was also accessed as a function of number of RHS, as seen in Figure 7. As discussed in Section 3, the solving of different RHS is serialized within the CUDA program. Likewise, the execution time scales linearly with number of RHS. This seems to also be true for the MKL program for high RHS. However, the slope of the MKL line is shallower which means that although both programs serialize the forward and backward substitution, MKL does a better job.

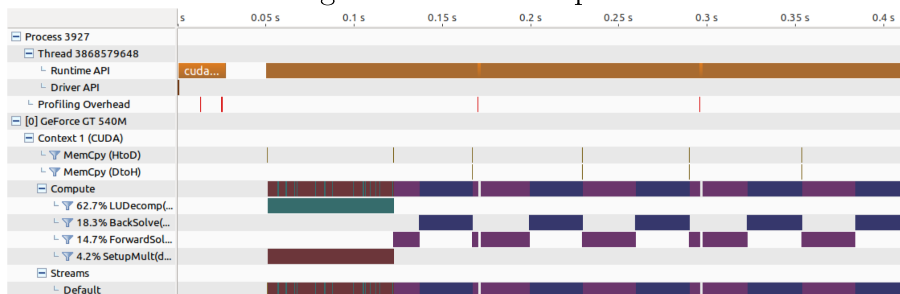
Figure 7: Right hand size scaling analysis. The matrix dimension was kept constant at 1000 with a bandwidth of 501.



4.2 Profiling with NVVP

NVVP was run with a matrix dimension 1000, a bandwidth of 501, and 10 RHS. Results are shown in Figure 8. This shows that even with a modest number of RHS, the forward and backward substitutions dominant the execution time. Since the solving of each column in the b matrix is independent, the implementation could exploit further parallelization. This would result in a speedup of approximately 2.

Figure 8: NVVP output.



4.3 CUDA memcheck

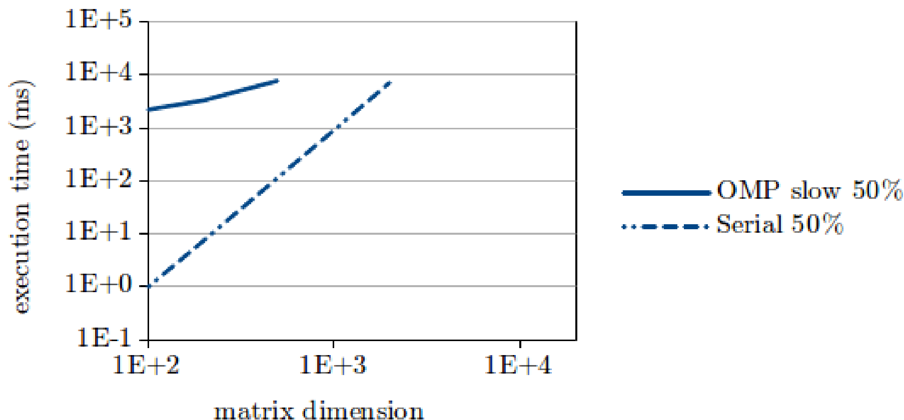
CUDA memcheck was run using a matrix dimension of 1000, a bandwidth of 501 and 10 RHS. Full output appears in Appendix A. The memcheck raises two incidences of ”Program hit error 17 on CUDA API call to cudaFree.” It is not clear where this error is coming from. This may indicate that an array is being over-stepped, but this is somewhat unlikely because the program gives correct results over a large range of dimensions and bandwidths without segmentation faults. One might suspect that this is arising from the use of pinned host memory, since two blocks of pinned host memory are used (for arrays A and b).

5 OpenMP

An attempt was also made to implement a banded system solver using OpenMP. This was done by first creating a serial solver using the algorithms described by Figures 1, 2, 3, on a matrix stored in banded format.

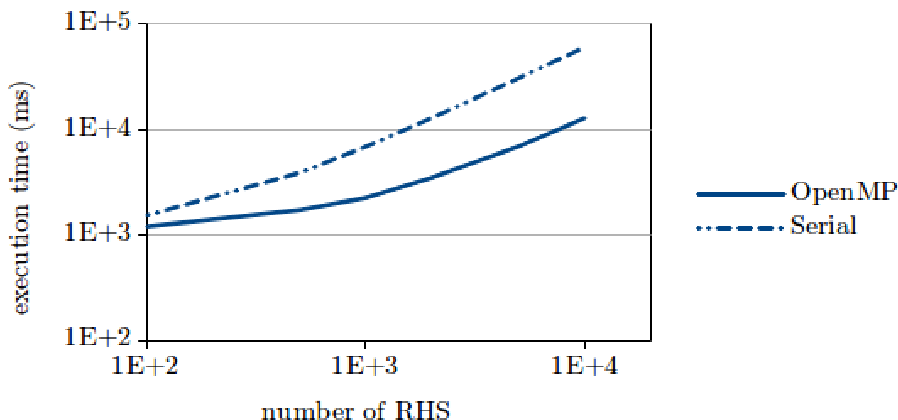
As it turns out, these algorithms do not offer much opportunity for parallelization using OpenMP. OpenMP threads have a considerably higher overhead associated with creation. This makes the tiling strategy used in CUDA programming difficult to replicate. For example, in the CUDA program, LU decomposition was done by employing blocks to fill an active region of the matrix. This same strategy was attempted with OpenMP by creating a parallel `for` loop (line 5 of Figure 1) with private index variables. This was done (in file `omp_solve_slow.c`), resulting in program that is considerably slower than the similar serial program. This is shown in Figure 9. The problem is that these inner `for` loops are executed $N - 2$ times by the outer for loop, each time with an expensive thread creation step. Since the axes of Figure 9 are the same as Figure 6 it is easy to verify that the CUDA and MKL programs are much faster than the serial program.

Figure 9: Scaling analysis for serial and slow OpenMP implementation for 50% bandwidth. Axes are the same as Figure 6.



The same problems associated with paralleling LU decomposition exist when parallelizing the forward/backward substitution. For this reason, it appears the only worthwhile way of parallelizing the serial algorithm is by parallelizing the forward/backward solve of multiple RHS. This was done in file `omp_solve.c`. The matrix dimension and bandwidth scaling for this program should be identical to the serial program. The RHS scaling is shown in Figure 10. This Figure shows that OpenMP offers an order of magnitude speedup over the serial program. The axes of Figure 10 are the same as 7. Comparing these figures revels that the OpenMP program is also considerable faster than the CUDA program, but still slower than the MKL program. In other words, the serial MKL program seems to scale better than even a fully parallel OpenMP implementation.

Figure 10: RHS scaling analysis for OpenMP and serial programs. The matrix dimension was kept constant 1000 with a bandwidth of 501. For OpenMP, the maximum number of threads was used: 16.



6 Conclusion

In this project, a CUDA program was created to solve a diagonally dominate banded system using an algorithm to operate on an A matrix stored in banded matrix form. The program is systematically faster than the Intel MKL banded matrix solver for a single RHS, but scales much less favorably for multiple RHS. NVVP results highlight the potential for improved parallelization of solving multiple RHS. The algorithm chosen for the CUDA implementation does not carry over well into the OpenMP implementation due to a higher overhead associated with OpenMP thread creation. For this reason, the only meaningful parallelization achieved was parallelizing the solving of multiple RHS. Though this method is proved to be faster than CUDA, the MKL program is still faster. In order to achieve better results with OpenMP, exploring entirely different algorithms may be necessary.

References

[1] D. Heath. Parallel numerican algorithms. http://courses.engr.illinois.edu/cs554/notes/06_lu_8up.pdf. CS554: University of Illinois at Urbana-Champaign.

7 Appendix A: CUDA full memcheck results

```
>> cuda-memcheck ./cuda_solve 11 3 1
Test PASSED
========= CUDA-MEMCHECK
========= Program hit error 17 on CUDA API call to cudaFree
=========     Saved host backtrace up to driver entry point at error
=========     Host Frame:/usr/lib64/nvidia/libcuda.so [0x30af90]
=========     Host Frame:./cuda_solve [0x44036]
=========     Host Frame:./cuda_solve [0x3a02]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1ed1d]
=========     Host Frame:./cuda_solve [0x2dd9]
=========
========= Program hit error 17 on CUDA API call to cudaFree
=========     Saved host backtrace up to driver entry point at error
=========     Host Frame:/usr/lib64/nvidia/libcuda.so [0x30af90]
=========     Host Frame:./cuda_solve [0x44036]
=========     Host Frame:./cuda_solve [0x3a0c]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1ed1d]
=========     Host Frame:./cuda_solve [0x2dd9]
=========
========= ERROR SUMMARY: 2 errors
```