Accelerating ‘fields’ by revamping the Cholesky Decomposition

Vinay B. Ramakrishnaiah

Raghu Raj P. Kumar

John Paige

Dorit Hammerling

NCAR Technical Notes
NCAR/TN-518+STR

National Center for Atmospheric Research
P. O. Box 3000
Boulder, Colorado
80307-3000
www.ucar.edu

NCAR TECHNICAL NOTES

http://library.ucar.edu/research/publish-technote

The Technical Notes series provides an outlet for a variety of NCAR Manuscripts that contribute in specialized ways to the body of scientific knowledge but that are not yet at a point of a formal journal, monograph or book publication. Reports in this series are issued by the NCAR scientific divisions, serviced by OpenSky and operated through the NCAR Library. Designation symbols for the series include:

EDD – Engineering, Design, or Development Reports
Equipment descriptions, test results, instrumentation, and operating and maintenance manuals.

IA – Instructional Aids
Instruction manuals, bibliographies, film supplements, and other research or instructional aids.

PPR – Program Progress Reports
Field program reports, interim and working reports, survey reports, and plans for experiments.

PROC – Proceedings
Documentation or symposia, colloquia, conferences, workshops, and lectures. (Distribution maybe limited to attendees).

STR – Scientific and Technical Reports
Data compilations, theoretical and numerical investigations, and experimental results.

The National Center for Atmospheric Research (NCAR) is operated by the nonprofit University Corporation for Atmospheric Research (UCAR) under the sponsorship of the National Science Foundation. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

National Center for Atmospheric Research
P. O. Box 3000
Boulder, Colorado 80307-3000

NCAR/TN-518+STR
NCAR Technical Note

2015-08

Accelerating 'fields' by revamping the Cholesky Decomposition

Vinay B. Ramakrishnaiah
Dept. of Electrical Engineering,
University of Wyoming, Laramie, WY
Raghu Raj P. Kumar
Technology Development Division,
National Center for Atmospheric Research, Boulder, CO
Dept. of Electrical Engineering,
University of Wyoming, Laramie, WY
John Paige
Lawrence Berkeley National Laboratory,
Berkeley, CA
Dorit Hammerling
Institute for Mathematics Applied to the Geosciences,
National Center for Atmospheric Research, Boulder, CO

Computational and Information Systems Laboratory (CISL)
Institute for Mathematics Applied to the Geosciences (IMAGe)

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
P. O. Box 3000
BOULDER, COLORADO 80307-3000
ISSN Print Edition 2153-2397
ISSN Electronic Edition 2153-2400

Accelerating 'fields' by revamping the Cholesky Decomposition

Vinay Ramakrishnaiah, Raghu Raj Prasanna Kumar, John Paige, Dorit Hammerling [1]
July 7, 2015

Abstract
The Geophysical Statistics project group within the Institute for Mathematics Applied to Geosciences (IMAGe) has been making use of Matrix Algebra on GPU and Multicore Architectures (MAGMA) to accelerate the Cholesky decomposition. The acceleration is motivated by a) Its frequent use in key computations in the spatial statistics R 'fields' package, b) Major bottleneck in 'fields' package execution and c) Operations involving big matrices make it suitable for parallelization. The Cholesky Decomposition was accelerated last summer using the MAGMA library. However, the performance of the accelerated version on multiple GPUs was observed to be unconventional - a) Execution time on multiple GPUs was higher in comparison to single GPU execution and b) Deep copy and in-place algorithms had opposite impacts on performance when executed on one and multiple GPUs. Our CPU and GPU profiling, conducted this summer, explains the unconventional behavior observed in the multi-GPU executions. The profiling provided insight to further accelerate the Cholesky Decomposition hierarchically– a) accelerating the underlying C function, b) reducing the function call overhead in R and c) optimizing the R environment. We were able to optimize the code and the environment to get a speedup greater than 75x (single precision) and 65x (double precision) for large matrices. We also found a potential way to improve the MAGMA functions by replacing the communications with direct device-to-device calls.

Keywords: High-Performance Computing, GPU, MAGMA, Cholesky Decomposition, fields, spatial statistics, Kriging.

[1] Vinay Ramakrishnaiah, is PhD student, University of Wyoming, 1000E University Ave. Laramie, WY 82070 (vramakr1@uwyo.edu), Raghu Raj Prasanna Kumar, is Associate Scientist II, National Center for Atmospheric Research, PO Box 3000, Boulder CO 30307-3000 (raghuraj@ucar.edu), John Paige, is Research Associate, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley CA 94720 (paigejo@uw.edu), Dorit Hammerling, is Project Scientist II, National Center for Atmospheric Research, PO Box 3000, Boulder CO 30307-3000 (dorith@ucar.edu).

Contents

List of Figures

List of Tables

1. INTRODUCTION

The ‘fields’ package for R is for spatial statistical computations. The major methods include cubic and thin plate splines, Kriging and compact covariances for large data sets. These methods are computationally complex and it becomes difficult to operate on huge data sets because currently the ‘fields’ package is implemented as a single thread version on CPU. Addressing such problems involves utilizing the resources such as hardware, time, money etc., which are usually limited, efficiently.

In recent times, computational devices come with very high clocking speeds for blazing fast calculations. We can make use of a powerful computer (like Yellowstone), which has dedicated hardware for handling and operating on very large data sets. According to Moore’s law: The number of transistors per square inch in an integrated circuit doubles approximately every two years. But as one could guess, Moore’s law is reaching its limit as the number of transistors cannot keep doubling forever and heat management becomes a tricky issue when running at very high clock speeds. So, people are switching to alternative methods to make the computations faster and parallel computing is one such method. In simple words, parallel computing is simultaneous use of multiple computing cores to solve a problem by breaking the problem to distinct concurrently solvable parts. In this way, complex problems requiring a lot of computational time can be accelerated.

2. MOTIVATION

In the previous work (Paige, Lyngaas, Ramakrishnaiah, Hammerling , Kumar, & Nychka, 2015), the authors discuss accelerating the ‘fields’ package by using Matrix Algebra on GPU and Multicore Architectures (MAGMA) (http://icl.cs.utk.edu/magma/) (Tomov, Dongarra, Volkov, & Demmel) and methods to use the accelerated functions in R. The results showed that the two-GPU implementation with deep copy was sometimes slower than the one-GPU implementation. Kriging is an interpolation technique in which the interpolated values are modeled by a Gaussian process governed by prior covariances. The Kriging algorithm has a computational complexity of the order of $O(n^3)$ and a memory complexity of the order of $O(n^2)$ for n spatial observations. The biggest computational problem is associated with the evaluation of the data likelihood function given the covariance parameters, which requires the Cholesky decomposition (Katzfuss and Cressie 2012). The MAGMA library provides a scalable and optimized method to compute the Cholesky decomposition.

3. METHODS AND INITIAL TIMING RESULTS

During the development process, we came up with two versions of code using the MAGMA library. An R wrapper is invoked when ‘magmaChol’ (an accelerated version of the R function ‘chol’ for the Cholesky decomposition) is called. The wrapper makes calls to the shared library built using an accelerated code, which computes Cholesky decomposition.

When ‘magmaChol’ is invoked with nGPU>1 (number of GPUs to use for computations),

it computes the Cholesky decomposition by calling MAGMA functions. MAGMA has two functions to compute the Cholesky decomposition on multiple GPUs, dpotrf_m and dpotrf_mgpu. dpotrf_m is a CPU function making GPU calls, handling multiple GPUs per node. dpotrf_m takes a positive definite matrix of dimension 'N', residing in the CPU memory as input, and computes the Cholesky decomposition. dpotrf_mgpu is a GPU function which operates on the positive definite matrix residing in the GPU memory. MAGMA also provides C programs for testing the functions 'dpotrf_m' and 'dpotrf_mgpu'. The following figure shows the comparison of the two functions as obtained when we run the MAGMA testing functions.

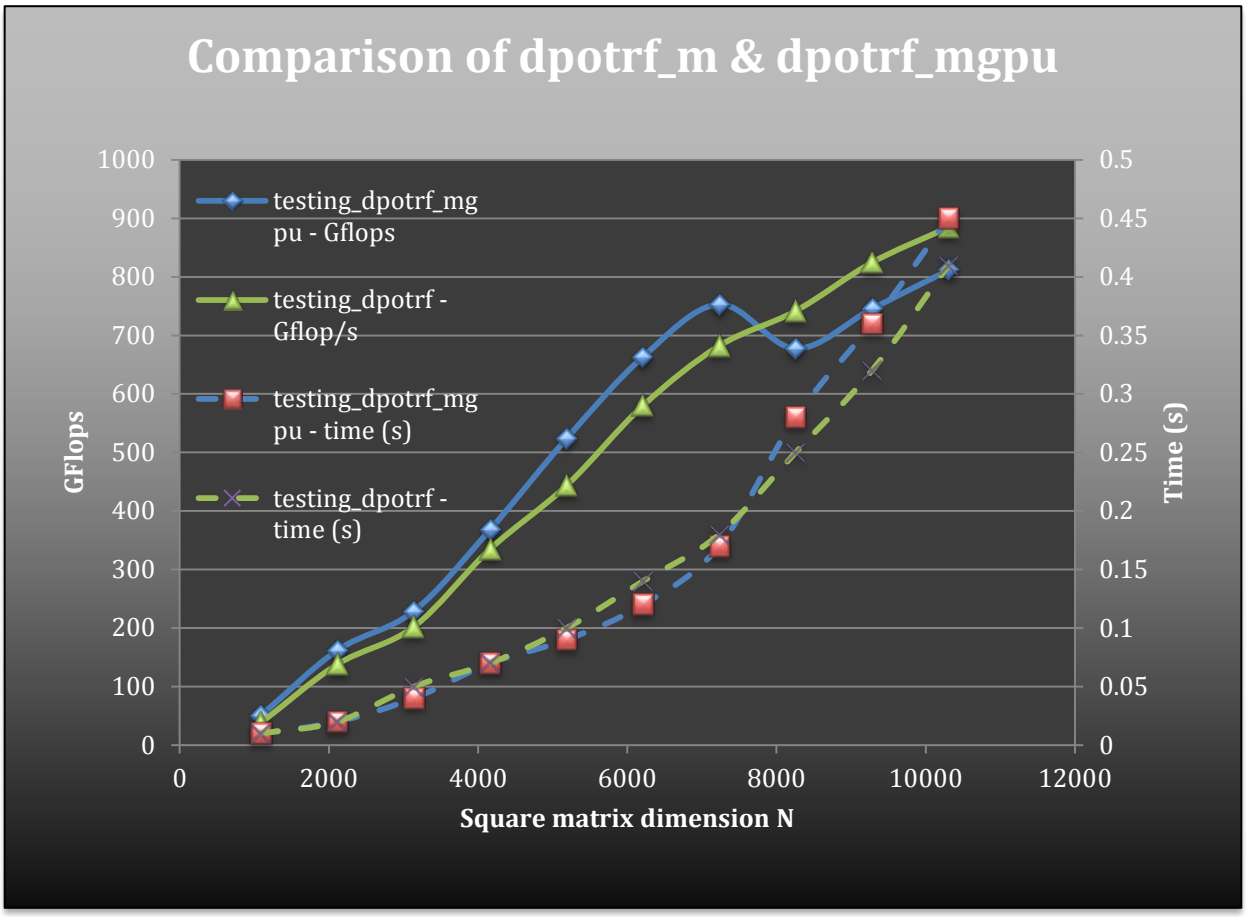


Figure 1: Graph comparing the performance of the testing programs – testing_dpotrf and testing_dpotrf_mgpu.

From Figure 1 we can see that the time taken for computations by the program testing_dpotrf_mgpu is very close (but slightly higher) for square matrices of dimension less than 82K when compared to the program testing_dpotrf. For matrices of dimension greater than 82K, the former program outperforms the later.

In order to analyze the code, we wrote our own testing function in C that feeds a positive definite matrix into the function dpotrf_m as input and we used the NVIDIA visual profiler for profiling the code. Here is the C code used for profiling:

```
#include <stdio.h>
#include <cuda.h>
#include <time.h>
#include "magma.h"

int main ( int argc , char ** argv )
{
    magma_init (); // initialize Magma
    clock_t start , end;
    double gpu_time ;
    magma_int_t info , i, j, k;
    magma_int_t  noOfRows  =  8192;  //  a  –  noOfRows  x  noOfRows
matrix
    magma_int_t noOfElements=noOfRows*noOfRows; // size of a
    double *a,*b; //
    int iterations=1;
    // allocate matrices on the host
     magma_dmalloc_cpu ( &a , noOfElements ); // host memory for
a
     magma_dmalloc_cpu ( &b , noOfElements ); // host memory for
a

    //initialze a positive definite matrix
    for(i=0; i<noOfRows; i++)
    {
            for (j=0; j<=i; j++)
            {
                    a[j*noOfRows+i]  =  (rand()%10)  /  100.0  +
0.1;
                    a[i*noOfRows+j] = (a[j*noOfRows+i]);
                    b[i*noOfRows+j]    =    b[j*noOfRows+i]    =
a[i*noOfRows+j];
            }
            a[i*noOfRows+i] += 100.0;//large diagonal elements
            b[i*noOfRows+i] = a[i*noOfRows+i];
    }
    gpu_time=0;
    for(i=0;i<iterations;i++)
    {

            start = clock();
            magma_dpotrf_m(2,MagmaLower,  noOfRows,  a,  noOfRows,
&info);
            end = clock();
            if(info != 0)
            {
                printf("Error in job!%d\n",info);
```

```
                goto terminate;
            }
            gpu_time  +=  ((double)  end-start)/(CLOCKS_PER_SEC);
// Magma  time
            for(j=0;j<noOfRows;j++)
                for( k=0;k<=j;k++)
                {
                        a[k*noOfRows+j]   =   a[j*noOfRows+k]   =
b[k*noOfRows+j];
                }
    }

    printf (" Elapsed Time : %g sec .\n",gpu_time*1.0/iterations
);
    terminate:
    free (a); // free host memory
    magma_finalize (); // finalize Magma
    return 0;
}
```

---

4. Performance Analysis

In order to accelerate a program, we have to analyze the underlying hardware architecture, the dataflow in the program, whether the program is utilizing all the hardware resources available, the execution time of the program, the memory bandwidth utilization in communication etc. We had to probe all these parameters and make modifications based on it to increase the performance.

4.1 Introduction to Yellowstone: All results in this report are obtained by executing the program on Caldera nodes of the National Center for Atmospheric Research's (NCAR's) Yellowstone (https://www2.cisl.ucar.edu/resources/yellowstone) supercomputing environment (Figure 2). The timing results were obtained as the mode over 20 iterations and the standard deviation was in the range [0.001, 0.01].

A single Caldera node has the following specifications:

- 64 GB DDR3-1600 memory per node (62 GB usable memory per node)
- IBM x360 M4, dual-socket nodes
- Two 8-core 2.6-GHz Intel Xeon E5-2670 (Sandy Bridge) processors per node with AVX
- FDR Mellanox InfiniBand, full fat tree
- 2 NVIDIA GPUs

- Tesla K20X
- Compute Capability 3.5
- 14 multiprocessors (192 cores each): 2688 total cores
- 732 MHz core frequency
- 6.0 GB of GDDR5 memory at 2.6 GHz, 384-bit bus
- ECC enabled for memory; 5.76 GB available to users
- PCIe Gen2 x16 host interface

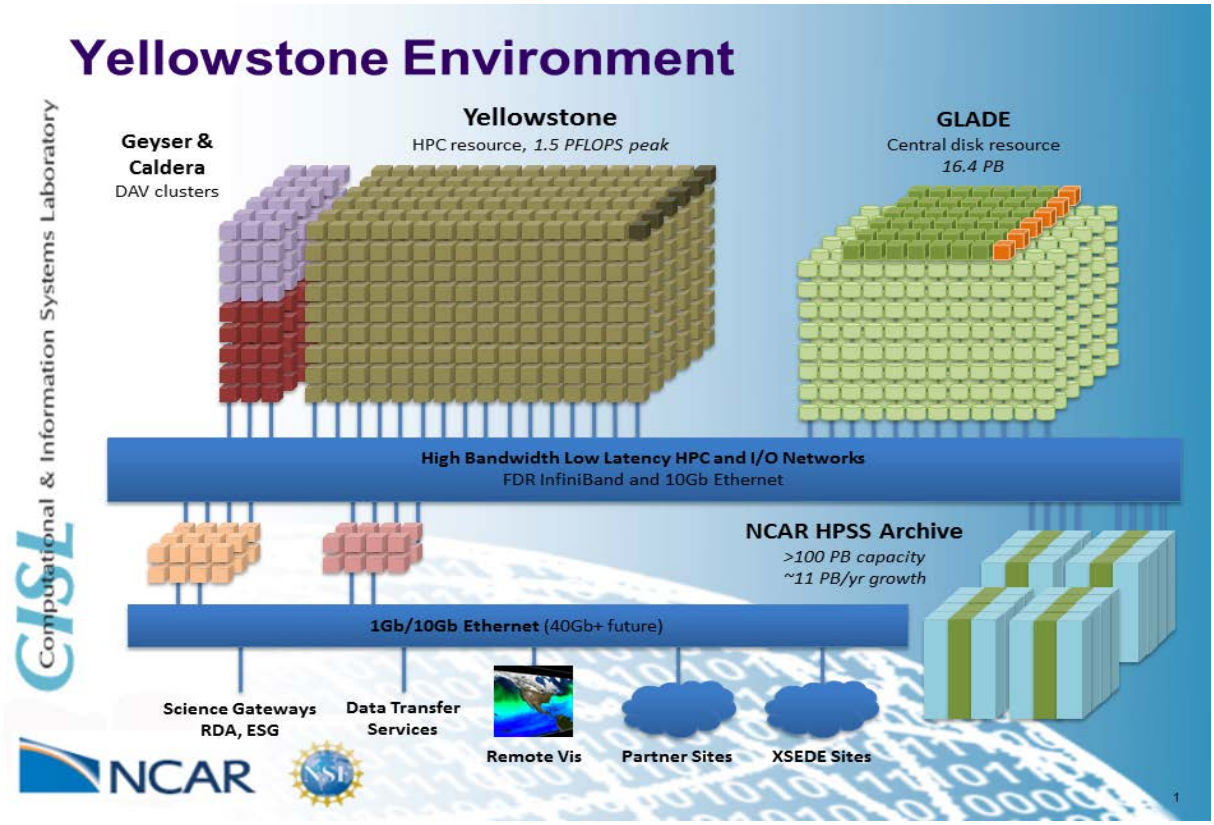
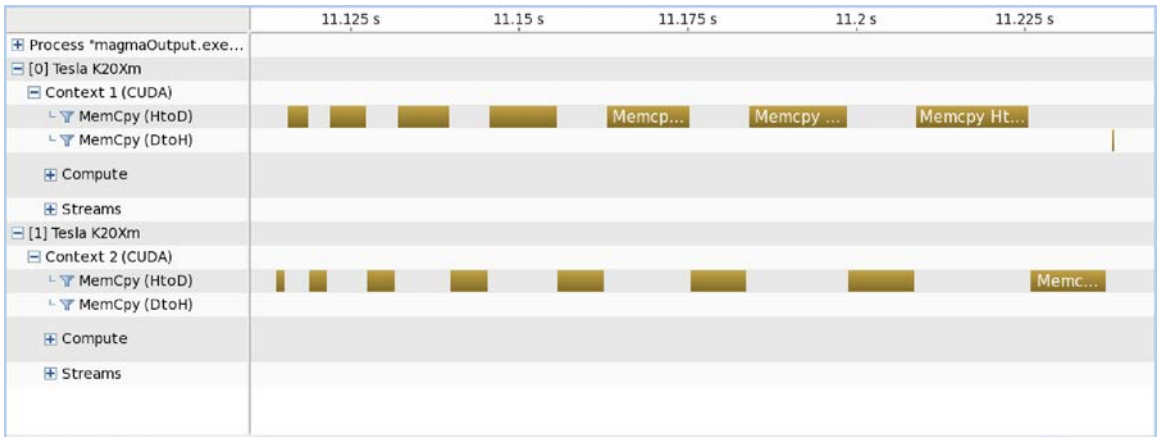


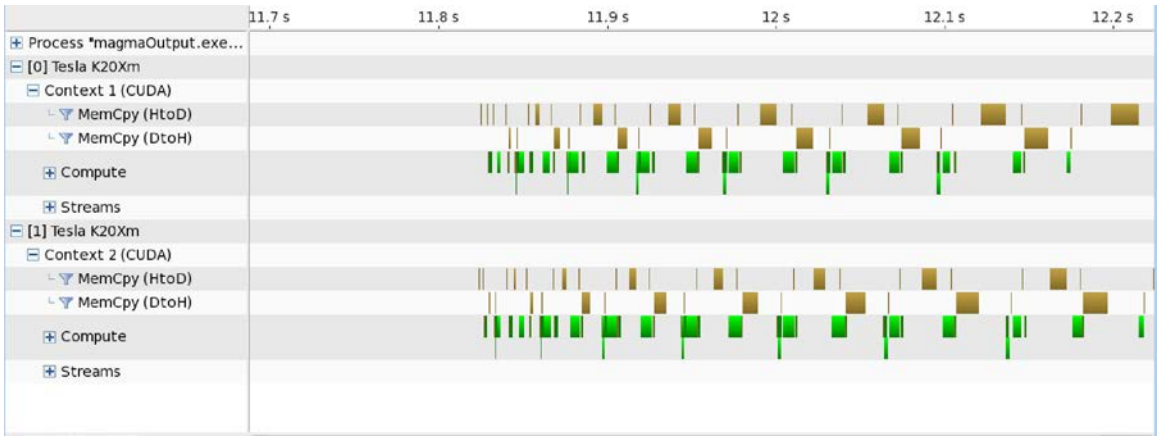
Figure 2: NCAR's Yellowstone Supercomputing environment. Figure shows the supercomputing, data storage, and archive systems. The GPU cards are on the Geyser and Caldera nodes indicated in Red.

4.2 Profiling results: Profiling is a form of dynamic program analysis where one can measure complexity, memory, time taken for particular instructions, frequency and duration of function call etc. The tool used for profiling is called a profiler. Since we were interested in memory transfers and usage of GPU, we used NVIDIA visual profiler as a profiling tool. The NVIDIA Visual Profiler (https://developer.nvidia.com/nvidia-visual-profiler) is a cross-platform performance-profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications.

Figure 3 shows the profiling result for nGPU=2 and leading dimension of the matrix 'N'=8192. The program was executed and profiled on the Caldera nodes.



(a)



(b)

Figure 3: Profiling results for the testing C program which calls dportrf_m. (a) Memory copy (b) Execution

While the configuration of Yellowstone super computer supports concurrent memory transfers from CPU to two GPUs, Figure 3 (a) shows that the initial memory copy from host to the two devices is sequential, i.e., when the data is transferred to one GPU, the other GPU does not receive any data. Further along the timeline (Figure 3(b)) we can see the computations and some data exchange (indicated in brown) between the GPUs. This data exchange is mainly due to the dependency of the algorithm on its previous steps. Also it is worthwhile to note that the data exchange between the two GPUs happens via the CPU (intermediate device to host communication in Figure 3(b)), while CUDA supports interdevice communication. If the hardware supports device-to-device communication, we can eliminate CPU from participating in the data communication, which will further increase the performance.

We profile both dpotrf (which computes Cholesky Decomposition) functions of Magma. When we profiled a testing program provided by magma to test the function dpotrf_mgpu, we obtained the following results.

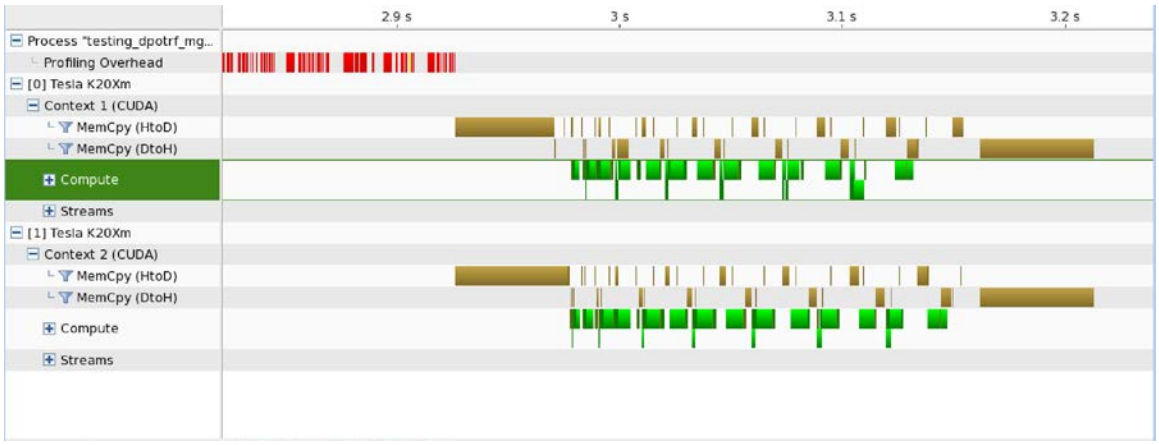


Figure 4: Profiling results for the testing C program which calls dportrf_mgpu

From Figure 4 we can see the asynchronous simultaneous transfer (indicated in brown) of data between the two devices. So we wrote a C program to include this function call and invoked it from the R wrapper. The C program used to call the function dpotrf_mgpu is as follows:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes, project
#include <R.h>
#include <Rdefines.h>
#include <cuda.h>
#include <cublas.h>
#include <magma.h>
#include <magma_lapack.h>
#include <time.h>

SEXP magmaCholeskyFinalNew_m(SEXP A, SEXP n, SEXP NB, SEXP
zeroTri, SEXP ngpu, SEXP lowerTri)
{
    magma_init();

    double *h_R;
    magmaDouble_ptr d_lA[ MagmaMaxGPUs ];

    magma_int_t N, n2, lda, ldda, max_size, ndev;
    magma_int_t info, nb;
    clock_t t1, t2;
    double gpu_time;
```

```
    int In, INB;
    In = INTEGER_VALUE(n);
    nb = INTEGER_VALUE(NB);
    double *PA = NUMERIC_POINTER(A);

    ndev = INTEGER_VALUE(ngpu);
    N=In;
    lda = N;
    n2 = lda*N;

    t1 = clock();
    nb = magma_get_dpotrf_nb(N);

     if ( MAGMA_SUCCESS != magma_malloc_pinned( (void**) &h_R,
(n2)*sizeof(double) )) {
        fprintf( stderr, "!!!! magma_malloc_pinned failed for:
%s\n", h_R );
        magma_finalize();
        exit(-1);
     }

    max_size = nb*(1+N/(nb*ndev)) * nb*((N+nb-1)/nb);
    for( int dev=0; dev < ndev; dev++ ) {
        magma_setdevice( dev );

        if ( MAGMA_SUCCESS != magma_malloc( &d_lA[dev],
(max_size)*sizeof(double) )) {
                fprintf( stderr, "!!!! magma_malloc failed for:
%s\n", d_lA[dev] );
                magma_finalize();
                exit(-1);
         }
     }
     t2 = clock();

     gpu_time = (double) (t2-t1)/CLOCKS_PER_SEC; // Magma time
     printf("Memory allocation time: %f seconds \n",gpu_time);


int lTri;
     lTri = INTEGER_VALUE(lowerTri);
     magma_uplo_t uplo;

       t1=clock();
       if ( lTri<=0 ) {
         uplo = MagmaUpper;
```

```
         ldda = ((N+nb-1)/nb)*nb;
// Transfer the matrix to device
         magma_dsetmatrix_1D_col_bcyclic( N, N, h_A, lda, d_lA,
ldda, ndev, nb );
       } else {
         uplo = MagmaLower;
         ldda = (1+N/(nb*ndev))*nb;
         magma_dsetmatrix_1D_row_bcyclic( N, N, h_A, lda, d_lA,
ldda, ndev, nb );
       }
t2=clock();
        gpu_time  =  (double)  (t2-t1)/CLOCKS_PER_SEC;  //  Magma
time
        printf("Memory transfer time: %f seconds \n",gpu_time);
    // Cholesky decomposition
       t1 = clock();
       magma_dpotrf_mgpu( ndev, uplo, N, d_lA, ldda, &info );
       t2 = clock();
       gpu_time = (double) (t2-t1)/CLOCKS_PER_SEC; // Magma time
       printf (" magma_dpotrf_mgpu time : %f sec. \n", gpu_time
);

           if (info != 0)
               printf("magma_dpotrf_mgpu  returned  error  %d:
%s.\n", (int) info, magma_strerror( info ));



        // Get the matrix from device
       if ( uplo == MagmaUpper ) {
         magma_dgetmatrix_1D_col_bcyclic( N, N, d_lA, ldda, h_A,
lda, ndev, nb );
       } else {
         magma_dgetmatrix_1D_row_bcyclic( N, N, d_lA, ldda, h_A,
lda, ndev, nb );
       }


      // Set the zero triangle

       int IZeroTri;
       In = INTEGER_VALUE(n);
       IZeroTri = INTEGER_VALUE(zeroTri);
       if(IZeroTri & lTri) {
              for(int i = 1; i < In; i++)
              {
                     for(int j=0; j< i; j++)
```

```
                        {
                                PA[i*In+j] = 0.0;
                        }
                }
        }

        else if(IZeroTri){
                for(int i = 0; i < In; i++)
                {
                        for(int j=i+1; j < In; j++)
                        {
                                PA[i*In+j] = 0.0;
                        }
                }
         }

        for( int dev=0; dev < ndev; dev++ ){
             magma_setdevice( dev );
             magma_free(d_lA[dev]);
        }

        magma_free_pinned(h_R);
        fflush( stdout );
        magma_finalize();

        return(R_NilValue);
}
```

4.3 Timing Results: Figure 5 compares the performance for the case when the two functions are invoked (separately). We can see that the function 'dpotrf_m' performs better than 'dpotrf_mgpu'. This contradicts our profiling results, which shows that asynchronous memory transfer in the function 'dpotrf_mgpu', resulting in overlap of communication and computation has a good raw speedup when compared to the CPU function making GPU calls (dpotrf_m).

Unfortunately, the function dpotrf_mgpu, which has a rather good performance as a raw C function, does not perform well when used with R. Different trials show that when the matrix is created on a pinned memory, it boosts the speed. Pinned memory is memory allocated using the *cudaMallocHost* function, which prevents the memory from being swapped out and provides improved transfer speeds. Non-pinned memory is memory allocated using the *malloc* function.

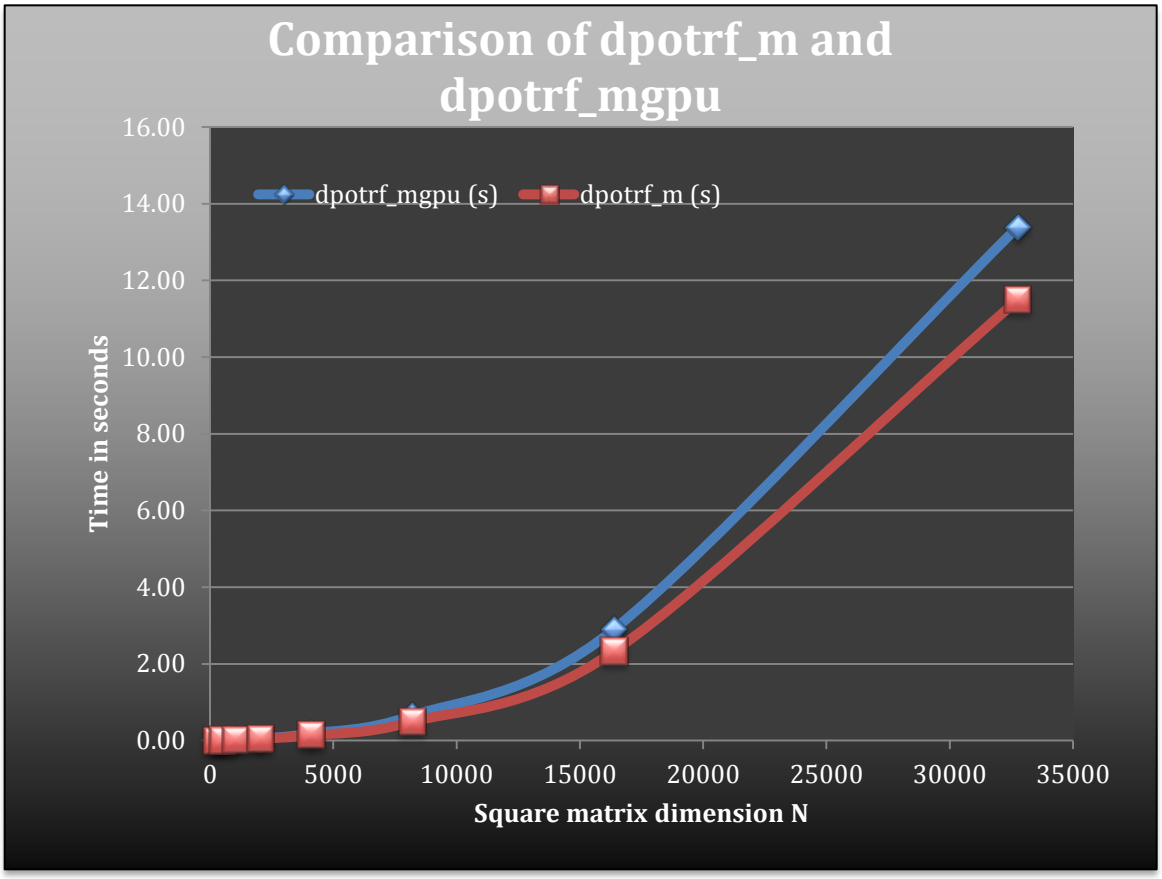


Figure 5: Graph comparing the two MAGMA functions when incorporated with R. The function 'dpotrf_m' is a CPU function making GPU calls and 'dpotrf_mgpu' is a GPU function.

As described in (http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/), which I am quoting - Host (CPU) data allocations are pageable by default. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or "pinned", host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory, as illustrated in Figure 6.

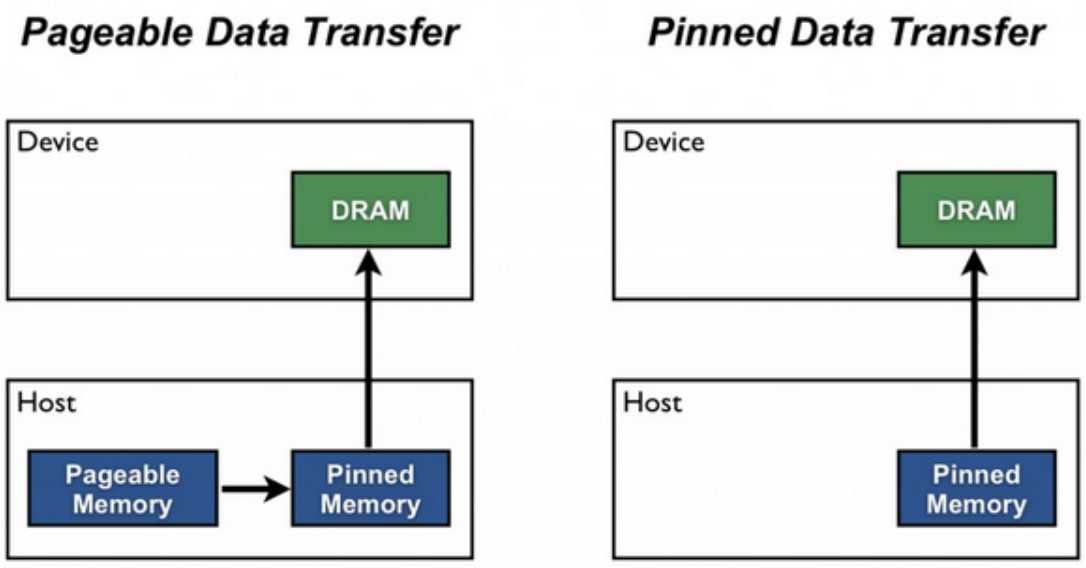


Figure 6: Figure showing the difference between non-pinned memory and pinned memory data transfer.

As can be seen in Figure 6, pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between page-able and pinned host arrays by directly allocating our host arrays in pinned memory. Pinned memory is much more expensive to allocate and de-allocate but provides higher transfer throughput for large memory transfers. For example, we used MAGMA's testing_dpotrf_mgpu program to test this with a square matrix of dimension 8192 and obtained the following results:

	GPU GFlops	Time taken for execution(s)
Pinned memory	820.13	0.22
Non-pinned memory	404.7	0.45

Table 1: With the matrix on pinned memory, the time taken for the Cholesky decomposition for a positive definite matrix of size 8192x8192 (double precision) with the number of blocks 'nb'=512 is 0.22s. Without having the matrix on pinned memory, the time taken for Cholesky decomposition for a positive definite matrix of size 8192x8192 (double precision) with the number of blocks 'nb'=512 is 0.45s which is a drop in performance by a factor of ~2.

But, since R does not have any facility to allocate pinned memory, we are losing performance. Even though a detour, we tried copying the matrix from host memory to pinned memory in C and then to the device memory, just to see whether the memory transfer overheads will reduce. We did not see any improvement in performance. Rather, it deteriorated because of the overhead involved in copying.

We have single precision version of Cholesky decomposition targeted towards fast computations on local machines. Since R always uses double precision, we had to force a copy to single precision in C in order to preform single precision computations. We used pinned memory for storing the single precision matrix, which resulted in saving computational time. This is more prominent when the number of GPUs used is more than one (2 in our case). This is because the allocation of pinned memory is slower than allocating non-pinned memory, but it will speed up the computations since there will be no memory swap. As 2 GPUs have better performance than

1 GPU, especially for large data, they can take advantage of the matrix on pinned memory. Figure 7 shows the timing results obtained for single precision calculations, when 2 GPUs were used for computations.

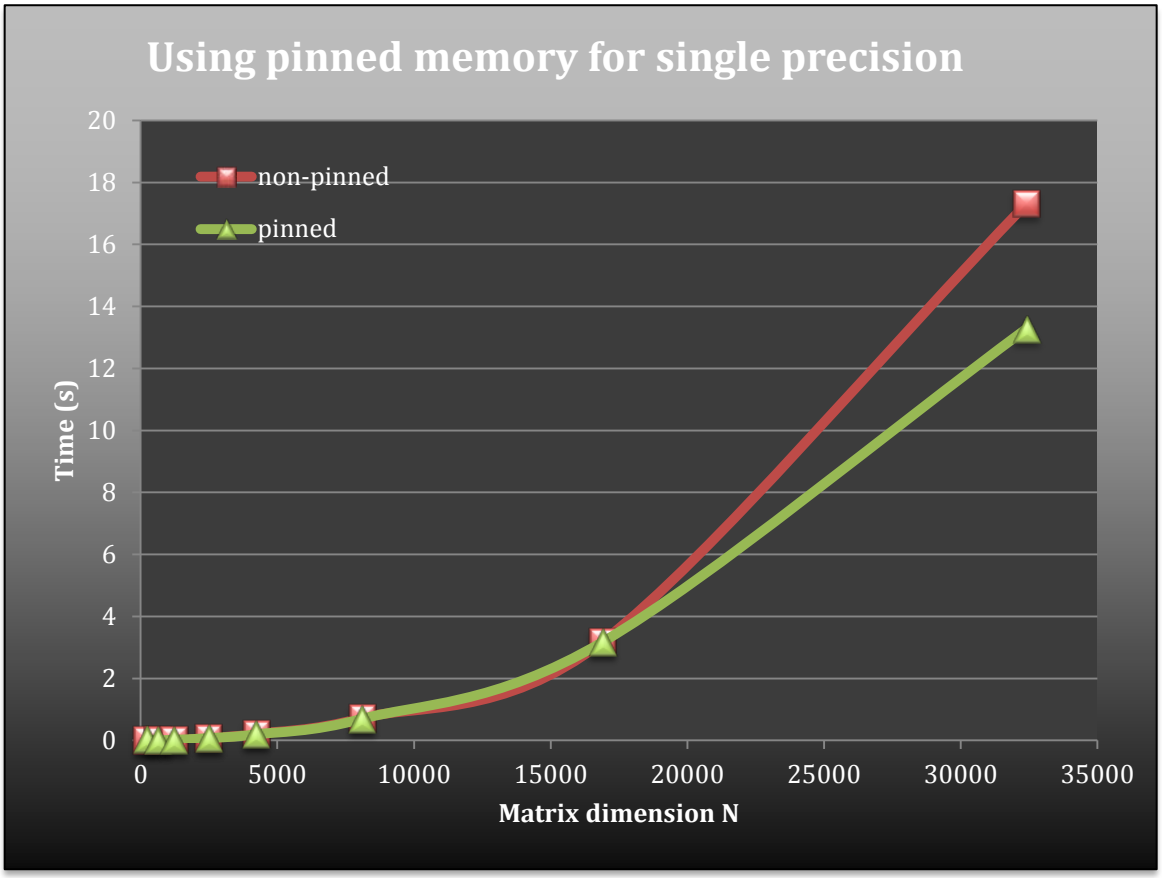


Figure 7: Graph showing the computation times for the single precision calculations with and without pinned memory.

5. Optimization Techniques

In order to include optimization on the compiler side we rebuilt MAGMA with the Intel 2015 compiler. This showed an improvement in overall performance especially for large matrix dimensions. Figure 8 shows the graphs comparing the performance of the code with MAGMA compiled using Intel 2012 and Intel 2015 compilers.

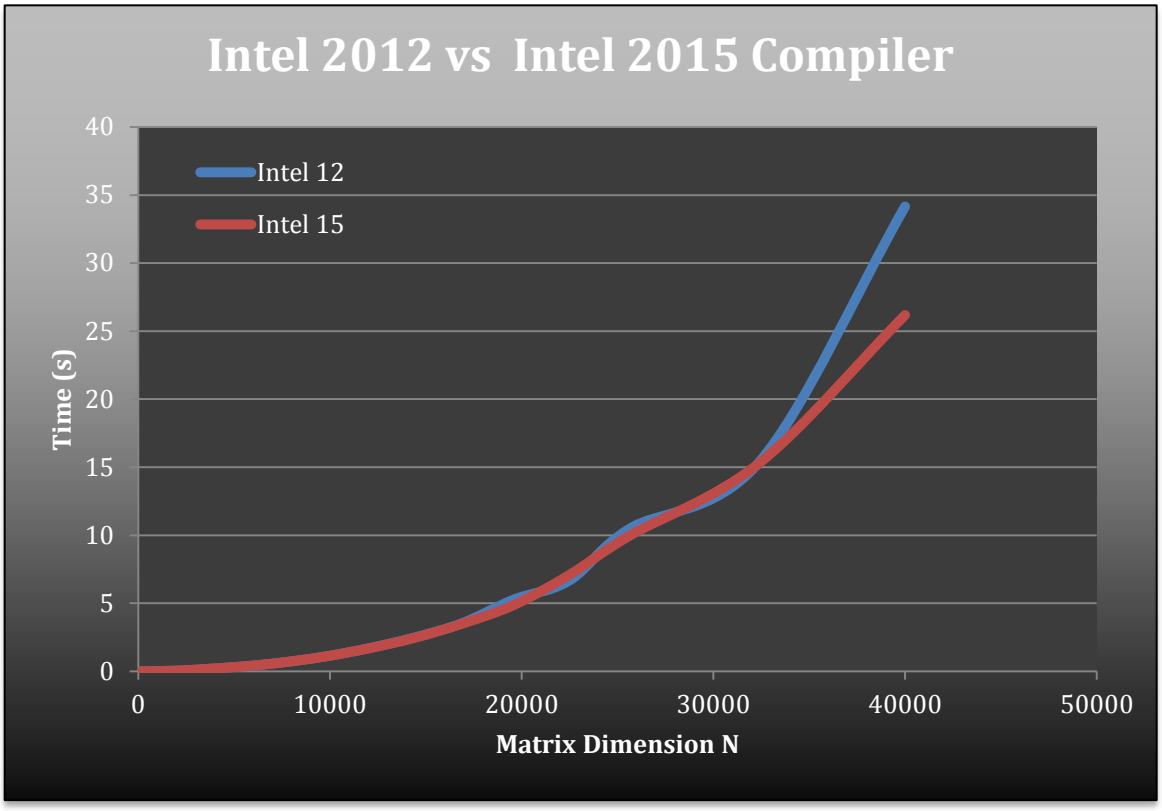


Figure 8: Graph showing the improvement in performance by the use of Intel 2015 compiler.

5.1 Overheads in R

We tried to get an estimate of the overhead involved by timing the function call 'magmaChol', the call to the C function from within the R wrapper using '.Call' and across the C function dpotrf_m and dpotrf_mgp. Figure 9 shows the overheads involved in the R calls.

We can see that the overheads for the cases when R calls 'dpotrf_m' and 'dpotrf_mgpu' are significant and grows as the size of the matrix increases. Also, since the overhead behaves similarly for both functions, it indicates that this is not because of the call to the C function but because of R's memory management. In order to validate this point, we also timed across the .Call function in R which calls the C functions. From Figure 9 we can notice that the time across .Call is almost the same as the time taken by the functions for computations. The current structure of the R call was as follows: The R program sources the script fieldsMAGMA.r, which invokes the wrapper. The wrapper dynamically loads the shared objects, which facilitates calls to C functions, which in turn uses MAGMA. We tried moving the dynamic loading to a level above the wrapper. This improves the performance. Since R loads the necessary environment objects when program counter navigates to different scripts, it acts as an overhead for function calls. Hence we see an improvement in performance when we move the dynamic loads to a script which is a level above the previous implementation. As seen from Figure 9, when the matrix size is large, N=32768 (double precision), and nGPU=2, we obtained an improvement of ~800ms.

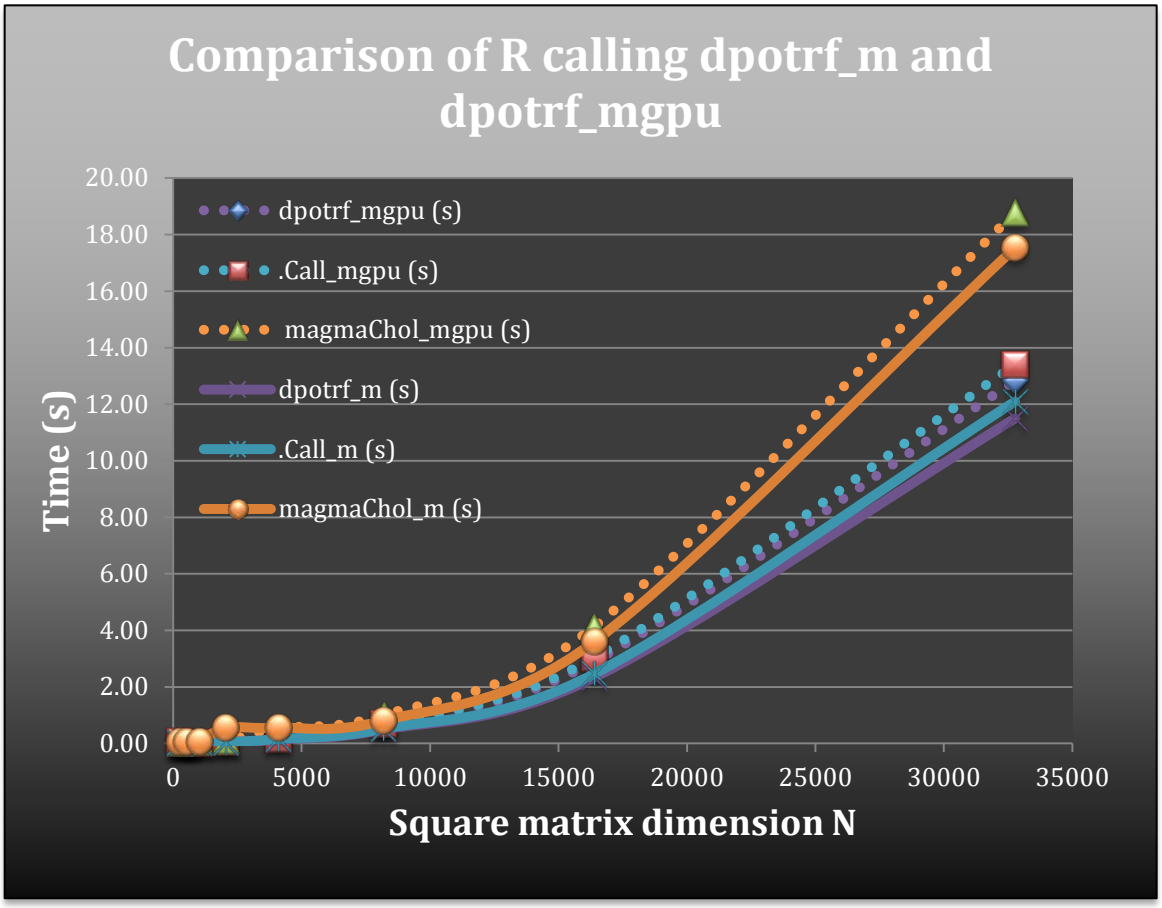


Figure 9: Graphs showing the overhead involved in R calls when the functions dpotrf_m and dpotrf_mgpu is called from R.

5.2 Intel LD_PRELOAD

This technique is used to quickly evaluate the performance impact of using optimized libraries without making any changes to the source code. According to Intel (https://software.intel.com/en-us/articles/optimizing-without-breaking-a-sweat), - "the Linux loader's LD_PRELOAD environment variable forces loading a shared library at run-time, allowing you to replace or extend the functionality of your application. The Linux loader does this by preempting functions in your applications to versions in the preloaded shared library. Functions from the preloaded shared library are inserted first into the Procedure Lookup Table (PLT), the global table of symbols for your application by the linker. This allows the replacement of globally bound, preemptable functions in your application, and is how we call highly optimized functions without any source or build system changes."

5.3 Single dynamic load

When a program or function is loaded into the Random Access Memory (RAM), only when it is necessary to be executed, it is called dynamic loading. Dynamic loading allows a program to proactively start in the absence of libraries, and gain additional functionality. Initially

we had multiple shared objects, which were dynamically loaded and were slowing down the code. We created a single shared object and called it with a single dynamic load, which improved the performance.

With preloading the Intel mkl shared libraries we got an improvement in performance in the range of [0.001s, 1s] and with the single dynamic load we were further able to achieve a small improvement in the range of [1ms, 800ms]. The improvement was greater for larger sized matrices. The results were obtained as an average over 20 iterations and the standard deviation was in the range [0.0001, 0.001].

6. ALTERNATIVE APPROACH

6.1 RCPP Package

The RCPP package developed by Dirk Eddelbuettel, provides C++ classes that facilitate interfacing C or C++ code in R packages. The package allows us to substitute C code for a function instead of R code, which accelerates the program since C is inherently faster. But the recognized problem with this approach is that, even though this approach is faster, having a precompiled shared library is much faster and RCPP also has an approach to compile shared libraries. R provides interface to C functions via '.Call()' function. We can call the C function by referring to the precompiled shared library. Since we were already using the .Call () interface provided by R for interfacing with C, we did not pursue the inline code approach.

7. SUMMARY OF RESULTS

Figure 10 shows the timing results summarized for all cases discussed in this report. Comparing the accelerated code with the CPU code shows the massive improvement in performance.

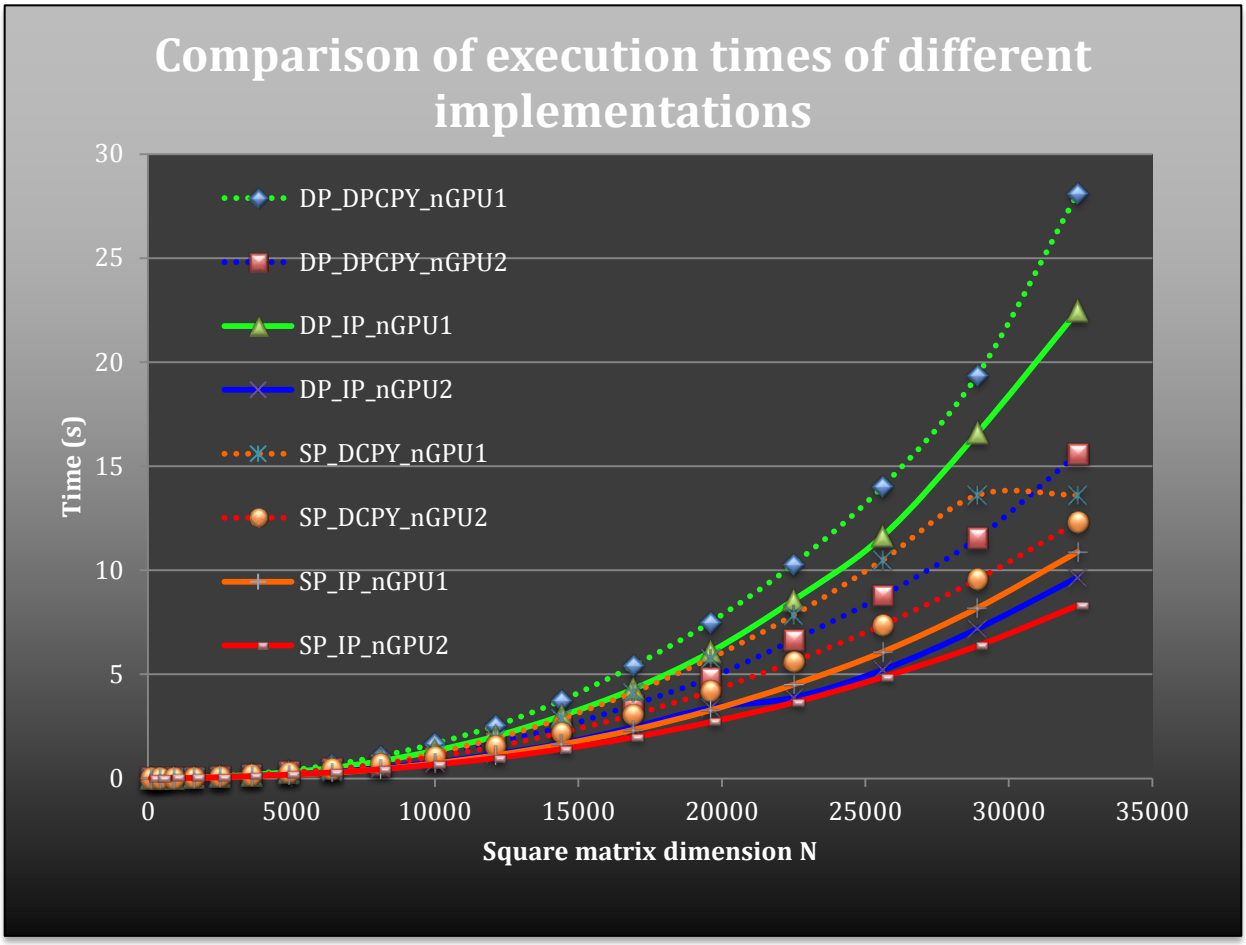


Figure 10: Graph showing the timing result for various cases. The abbreviations used are: SP – Single Precision; DP – Double precision; dpCPY – with deep copy; nGPU – number of GPUs used for computation.

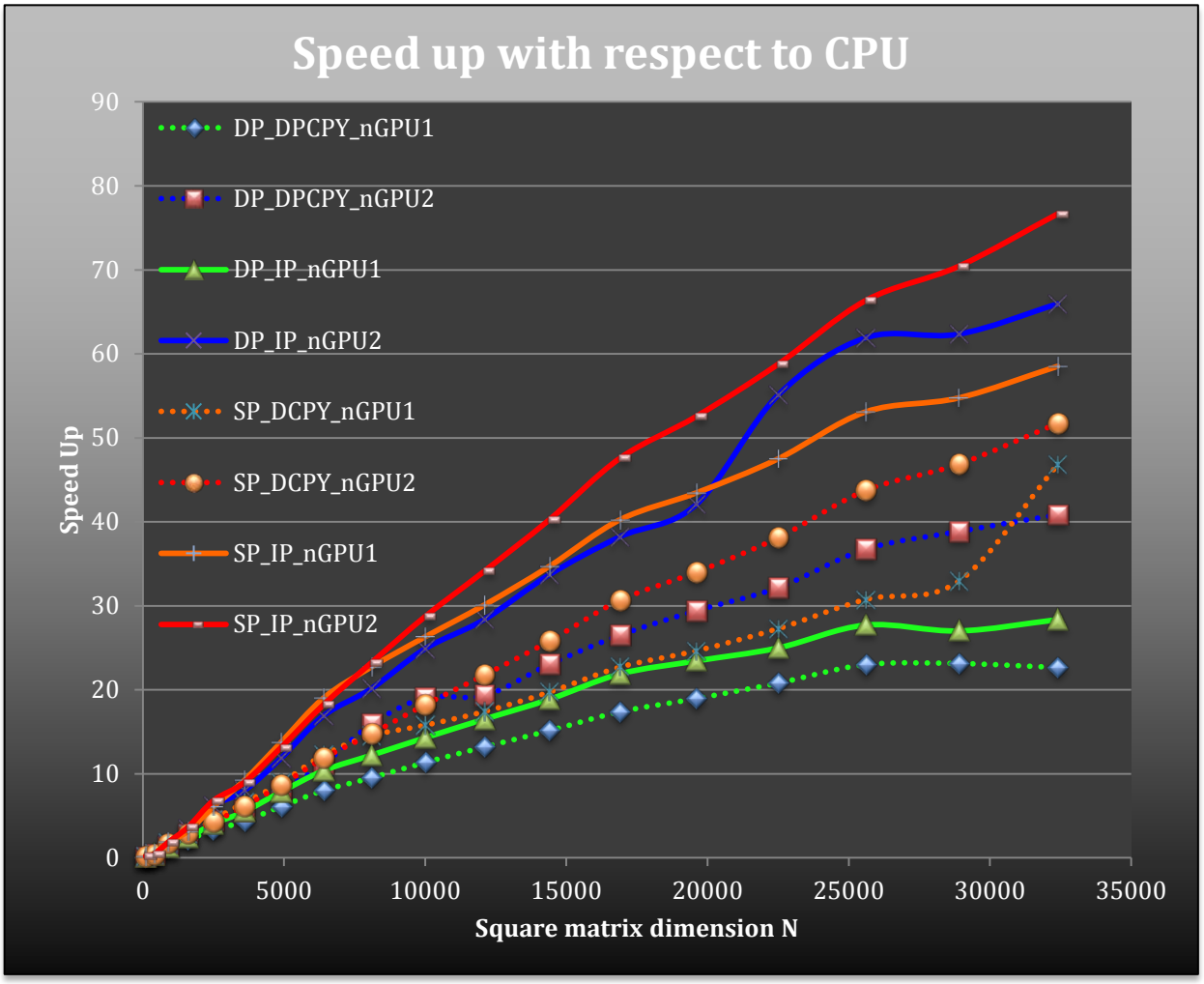


Figure 11: Graph showing the speed up of different GPU versions when compared to CPU.

By looking at Figure 10 and Figure 11 we can see some unconventional behavior of the graphs. In order to clearly understand the timing results and the behavior of the graphs in Figure 10 and Figure 11 we profiled the code using the performance analysis tool 'perf' in linux. We looked into the number of page faults, last level cache misses and data translation look-aside buffer (dTLB) misses to understand the performance of R and the C code on CPU. The number of Page faults give us an idea about the number of times the program could not find the data it was looking for readily available in the RAM, the number of cache misses indicates how effectively the program uses cache memory and the TLB misses shows how effectively the mapping is, between virtual addresses and physical addresses which gives us an idea about the interaction of the C program with R. Table 3 gives the results of profiling using 'perf'.

Page Faults (1e3)				
N	DC, 1GPU	DC, 2GPU	1GPU	2GPU
8192	65.177	82.212	64.091	81.593
16384	67.151	84.652	65.686	83.194
24576	73.528	88.921	69.383	87.455
32768	80.518	97.482	75.359	91.349
LLC Misses (1e6)				
N	DC, 1GPU	DC, 2GPU	1GPU	2GPU
8192	157.6	133.4	143.8	119.7
16384	505.1	402	451.8	394
24576	1059.7	823	955.3	715
32768	1821	1414	1650.7	1222
dTLB Misses (1e6)				
N	DC, 1GPU	DC, 2GPU	1GPU	2GPU
8192	7.8	6.6	8	6.5
16384	11.8	8.4	11.6	7.6
24576	17.9	9.8	18	9.9
32768	22.6	12.7	23.4	12.7
HtoD time (msecs)				
N	DC, 1GPU	DC, 2GPU	1GPU	2GPU
8192	65.1	307	65.1	274.53
16384	224.38	945	224.5	945.54
24576	508.3	2166.55	509.11	2162
32768	1159.2	3806	1159.55	3802
DtoH time (msecs)				
N	DC, 1GPU	DC, 2GPU	1GPU	2GPU
8192	45.1	85	45.1	84.9
16384	169.23	366	169.3	365
24576	384.5	796	379.23	803
32768	2991.53	1422	2842.55	1419
Elapsed Execution time-5 trials average (secs)				
N	DC, 1GPU	DC, 2GPU	1GPU	2GPU
8192	0.88	0.5	0.89	0.5
16384	4.03	2.12	4.08	2.13
24576	10.95	5.37	10.74	5.38
32768	22.29	11.83	22.3	11.75

Table 2: Table showing the results of profiling the code using 'perf'

The number of page faults for 2 GPU case is higher than 1 GPU case. This is because of the way in which computations are performed in MAGMA. Since MAGMA uses hybrid

architecture, it has both CPU and GPU code for computations. Since Cholesky decomposition involves dependency of results from previous steps, there is an evident information exchange between the two devices with CPU acting as a channel. We can notice that the LLC (Last level cache) misses is higher for the one GPU case because of more computations being carried on the CPU. Also, we can notice that the number of page faults and LLC misses increase with deep copy. Since the amount of memory used is more in case of deep copy, the RAM has to swap more data resulting in more page faults. It is also worthy to note that the TLB misses are similar for cases of deep copy as well as in-place computations. This shows that the only problem is with the data being swapped in the RAM.

We used our accelerated version of Cholesky decomposition for Kriging and obtained an improvement in performance (

Figure 12).

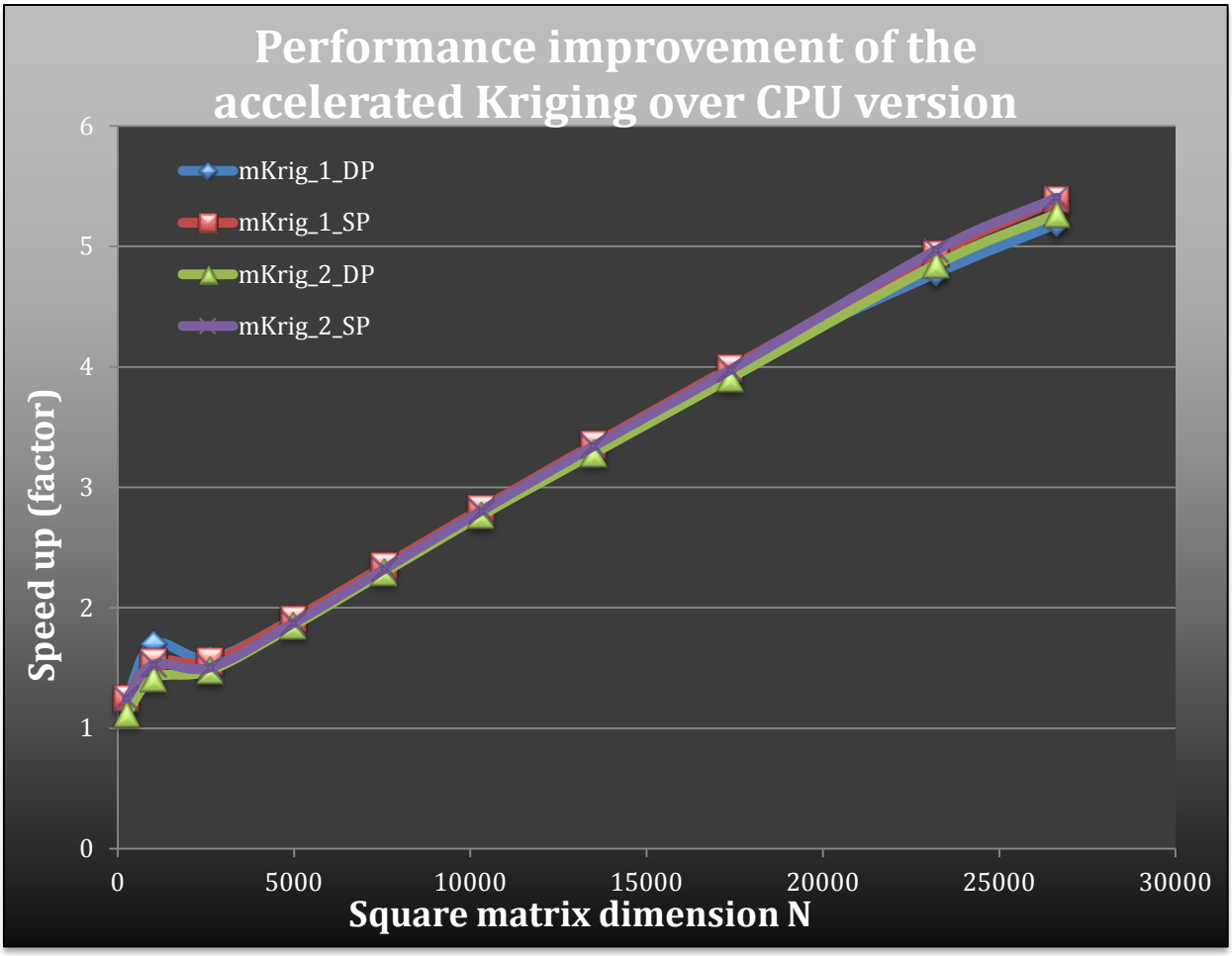


Figure 12: Bar graph showing the improvement in performance when accelerated versions of Cholesky Decomposition are used for Kriging.

Also, we tested our accelerated functions on the Kriging workflow and obtained an improvement in performance. Figure 13 shows the speed up obtained by making use of the accelerated version of Cholesky Decomposion for computing Kriging Workflow.

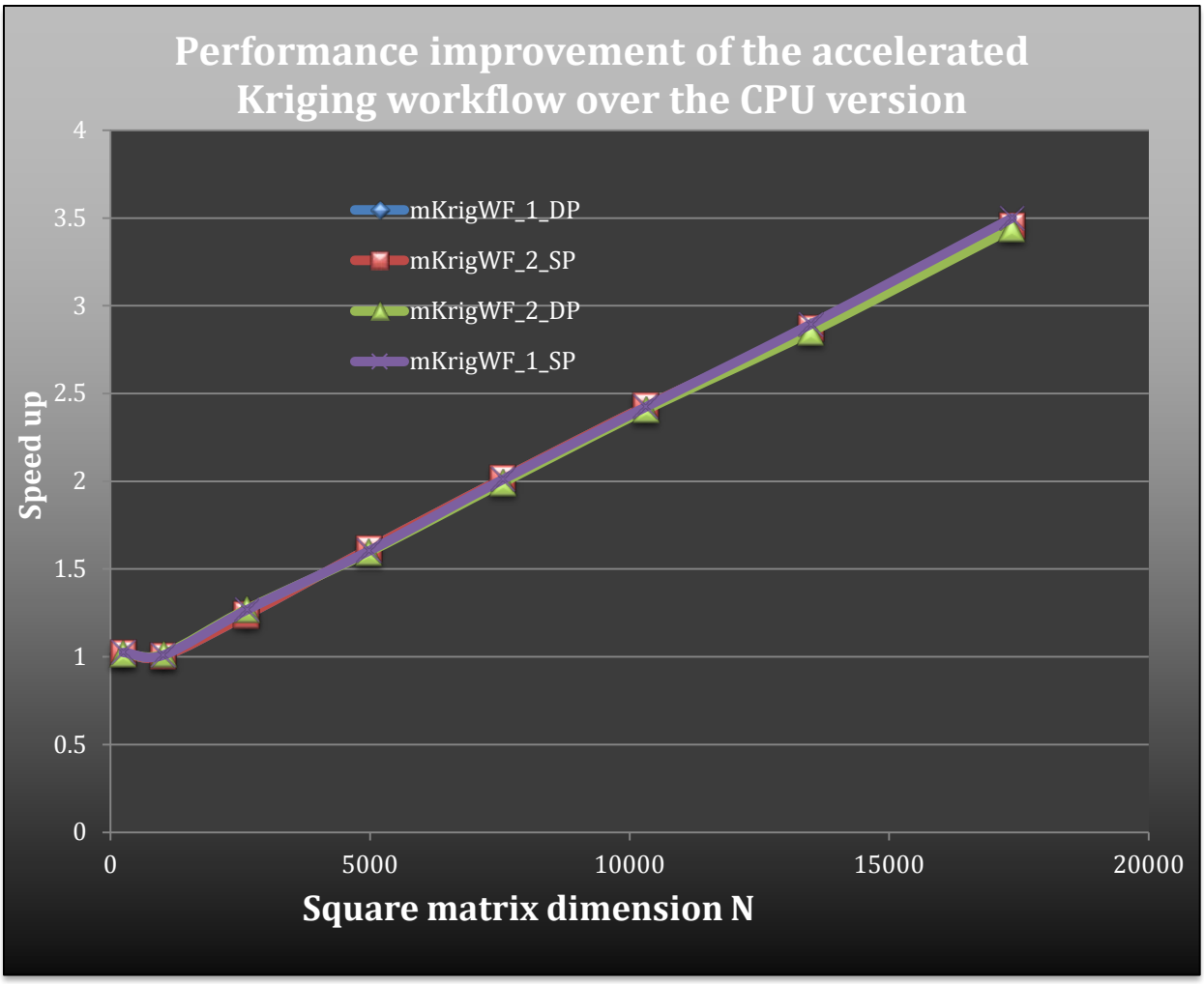


Figure 13: Bar graph showing the improvement in performance when accelerated versions of Cholesky Decomposition are used for computing Kriging workflow.

8. Conclusion and future work

The computation of the Cholesky decomposition in the 'fields' package was accelerated using the MAGMA library and improved by the methods specified in section 5. As seen from the results, the ability to allocate pinned memory can make a significant difference in performance. The overhead in R due to management of memory and environment is also large when compared to the accelerated computation times. In case of single precision with 2 GPUs, we obtained a significant speed up when pinned memory was used for copying the double precision matrix from R into a single precision matrix in C. We could think of further accelerating the Cholesky decomposition by modifying the MAGMA code. From the profiling results (Figure 3), we can see that the two GPUs are not directly exchanging data. We can eliminate intermediate communication with the CPU by having device-to-device data transfers. Also, there may be other provisions and room for fine adjustment, but that involves rewriting MAGMA code or writing a scalable Cholesky decomposition in CUDA C from scratch.

Bibliography

Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., et al. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA projects. *Journel of Physics: Conference series , 180* (012037).

http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/. (n.d.).

https://software.intel.com/en-us/articles/optimizing-without-breaking-a-sweat. (n.d.).

Katzfuss, M., & Cressie, N. (2012). *Bayesian hierarchical spatio-temporal smoothing for very large datasets* (Vol. 23). Environmetrics.

Paige, J., Lyngaas, I., Ramakrishnaiah, V., Hammerling , D., Kumar, R., & Nychka, D. (2015). *Incorporating MAGMA into the 'fields' spatial statistics package.* Boulder: UCAR/NCAR.

Tomov, S., Dongarra, J., Volkov, V., & Demmel, J. *MAGMA library.* University of Tenessee and University of California, Knoxville, TN, and Berkeley, CA.