Matrix multiplication on multiple GPU and FPGA server

Contents

[1 Table of Figures 2](#_Toc433466144)

[2 Abstract 2](#_Toc433466145)

[3 Background 3](#_Toc433466146)

[3.1.1 Introduction to the problem 3](#_Toc433466147)

[4 Project Goals 4](#_Toc433466148)

[5 Steps to solution 4](#_Toc433466149)

[5.1 MATLAB integration 4](#_Toc433466150)

[5.2 Using the GPU via MATALB 4](#_Toc433466151)

[5.2.1 Using cuBLAS 5](#_Toc433466152)

[5.2.2 Using streams 6](#_Toc433466153)

[5.3 Utilizing the CPU and GPU 8](#_Toc433466154)

[5.3.1 Using cuBLAS-XT 8](#_Toc433466155)

[5.3.2 Using the CPU 9](#_Toc433466156)

[5.4 Using CPU GPU and FPGA 10](#_Toc433466157)

[5.4.1 Using the FPGA 11](#_Toc433466158)

[5.4.2 Final integration 13](#_Toc433466159)

[6 Results 14](#_Toc433466160)

[7 Concluding remarks 16](#_Toc433466161)

[8 Table or meanings 17](#_Toc433466162)

[9 Code documentation 17](#_Toc433466163)

[9.1 Directories 17](#_Toc433466164)

[9.2 Files 17](#_Toc433466165)

[9.2.1 Main file MatrixMulMex\_1\_0.cu 18](#_Toc433466166)

[10 Installation process 19](#_Toc433466167)

[10.1 Regular use 19](#_Toc433466168)

[10.2 Install to compile 20](#_Toc433466169)

[10.3 FPGA integration 20](#_Toc433466170)

[10.4 Using Nsight 21](#_Toc433466171)

[11 Bibliography 21](#_Toc433466172)

# Table of Figures

[Figure 1: cuBLAS GPU calculation vs CPU calculation 6](#_Toc433464085)

[Figure 2: Streams using regular loops 7](#_Toc433464086)

[Figure 3: Open BLAS vs MATALB BLAS 9](#_Toc433464087)

[Figure 4: Streams compared to CuBLAS-XT 10](#_Toc433464088)

[Figure 5: Step 1, divide workload between CPU/GPU 11](#_Toc433464089)

[Figure 6: step 2 divide workload between FPGA and CPU 11](#_Toc433464090)

[Figure 7: Leading dimension explained 12](#_Toc433464091)

[Figure 8: CuBLAS-XT CPU/GPU compared to MATLAB 14](#_Toc433464092)

[Figure 9: Varying CPU usage percentage 14](#_Toc433464093)

[Figure 10:Nsight of final version 15](#_Toc433464094)

# Abstract

The purpose of this project was to explore the possibility of multiplying matrices in a parallel fashion via a user friendly function for MATLAB. The project explores the best ways to use heterogeneous systems to improve on MATLAB ability to perform very large scale matrix multiplications. The project was carried out on a desktop computer with 2 GPU’s and a multicore CPU, with an option to also integrate an FPGA.

# Background

### Introduction to the problem

There has always been a need to multiply matrixes as fast as possible. A few methods exist to multiply special matrixes (e.g. sparse matrixes). Our problem was to make the best use of all the computation resources available on a single pc, to calculate the product of 2 random floating point matrixes.

Introduction to hardware components.

* GPUs

Most modern GPUs consist of many small cores, great for executing small simple calculation, many times in parallel. Many PCs have discrete GPUs and we can harness this computing power to solve our problem. There is a limitation to calculating embarrassingly parallel problems on the GPU, and that is the data must be first transferred to the GPU and then transferred back once calculated.

* FPGA  
  A field programmable gate array can be specially tailored to any task, and use hardware to accelerate a calculation.
* CPU  
  Traditionally most calculation work is done on the CPU. The CPU is built to handle complex operations and might not be the most efficient way to “crunch numbers”. On the other hand, we can use it to distribute and manage calculations executed on other hardware available on the PC.
* MATLAB

MATLAB is commonly used to perform various calculation including, among others, matrices operations. It makes sense to try and enhance the capabilities of MATALB in multiplying matrices. In addition, MATALB gives many convenient tools to test the performance of our solution.

# Project Goals

To calculate level 3 BLAS [1] tasks on multiple computation platforms, that include Multi-Threaded CPU, GPUs, FPGAs in order to achieve the fastest possible performance on any machine with said tools.

# Steps to solution

## MATLAB integration

The first step to the solution consisted of integrating other coding languages and run them as an integral part of MATLAB. MATLAB has extensive documentation on how to integrate C/C++ files into MATLAB using the mex compiler [2]. This example was used to create the most basic C code and execute it from a MATLAB function [3].

## Using the GPU via MATALB

Our project uses Nvidia graphics to try and accelerate the process of matrix multiplication. Nvidia provides its own C like programming language to enable GPU specific programming [4] called CUDA. CUDA enables to create kernels which run on multiple cores and SM’s.

MATALB documentation has examples for using CUDA based files, as part of mex files for simple kernels [5]. These kernels give the programmer the ability to control the number of threads and blocks passed to the kernel, and optimize the code to best fit the matrices multiplied. Implementing the most straight-forward kernel implementation showed some improvement over CPU performance, but speed-up was not significant.

In order for the GPU to calculate the data, it must first be transferred to the GPU’s memory, and cannot calculate the data straight from the system memory. This means the way to performing a calculation is

* Allocate memory on the device
* Transfer data from host to device. .
* Perform the calculation by invoking a kernel. .
* Store the result on the device memory.
* Transfer the result from device to host.
* Read the result from the host system memory and free the device memory.

These steps could be performed over varying sizes of data. Once the data is allocated, it is calculated in a resolution of block and threads. The user can determine these parameters to best suit his hardware and software. Further documentation on determining the optimal number of blocks and threads could be found in Nvidia documentation[4].

The need to transfer the data to and from the GPU, means there is always time consumed for transferring the data, which might make it not efficient to perform GPU calculations.

This could be summarized in the following formula:

GPU calculation time =

Performing the calculation on the CPU requires no transfer time.

CPU calculation time =

Naturally, for very large matrices, GPU calculation gives more speedup as becomes the most dominant part of the total calculation time. Given that , it is in the best interest to perform these calculations on the largest matrices available.

### Using cuBLAS

Nvidia provides its own BLAS libraries, optimized for GPU linear algebra calculations called cuBLAS. CuBLAS libraries handle invoking the kernel themselves and could be easily integrated with the provided API[4]. CuBLAS libraries allow for better optimization of the block and thread size, dependent on the GPUs architecture and specifications. Using cuBLAS libraries makes it easier to integrate the software on any platform, without in-depth testing of each platform before use. Furthermore, performance on a given card may vary when using matrices of varying sizes. One blocks and threads configuration might be good for a certain matrix size and bad for another. CuBLAS determines these variables with no need for the developer to intervene.

Integrating cuBLAS libraries in our implementation has shown improvement over both raw CPU performance and the naïve kernel implementation. As expected, performance was relatively better when calculating larger matrices.



Figure 1: cuBLAS GPU calculation vs CPU calculation

*In the figure we see that there is a clear performance speedup when calculating matrix multiplications. The speedup becomes more substantial when larger matrices are multiplied.*

### Using streams

According to Nvidia, stream could be used on cards with compute capability of 1.1 or greater, to create multiple threads. This way, it would be possible to handle both data transfer between host and device, while still performing calculations on the device[6]. To achieve that, a slight different approach had to be applied to the matrices being multiplied. Each stream would multiply only part of the final result, therefore only part of each source matrix needed to be sent on each stream. To achieve this effect with the best results, each source matrix was grouped to a larger array so that a whole memory block could be sent with a single memory pointer. Effectively, the end result matrix would be calculated by multiplying smaller parts of the source matrices multiple times on many streams.

This made it possible to send sections of the array using streams, and begin processing them on the device once the first section has been received. Same goes for sending back the calculated data to the host. Effectively, pipelining the process described at the first section.

As the GPU memory is limited to only a few GB, and data must be first transferred to the GPU for calculation, there was a need to allocate the minimum memory between the number of chunks the matrixes were broken down into and the number of streams working on calculation the results. Limiting the allocated memory to made it possible to process matrixes larger than what could physically fit on the GPU memory.

Two implementation of this method were used. The first uses two sets of loops. The outer loop was used to break the large matrix into smaller ones, able to fit on the available GPU memory. The 3 inner loops asynchronously received data, processed it and sent it back to the host.

#### A general overview of first implementation

//streams\_num: number of workers

//Iteration\_num: could be received as a parameter

int batch **=** iteration\_num**/**streams\_num**;**

**for** **(**int l **=** 0**;** l **<** batch**;** l**++){**

**for(**int i **=** 0**;** i **<** streams\_num**;** i**++){**

cublasSetMatrixAsync**(**A**);**

cublasSetMatrixAsync**(**B**);**

**}**

**for(**int i **=** 0**;** i **<** streams\_num**;** i**++){**

cublasSgemm

**}**

**for(**int i **=** 0**;** i **<** streams\_num**;** i**++)** **{**

cublasGetMatrixAsync**(**C**)**

**}**

**}**



Figure 2: Streams using regular loops

*This graph is generated by Nsight* [7]*. This allows us to track which method and which process is in use. In green we can see the data is being sent to the device, in this case 4 streams were created to handle data transfer. In red and purple the data is being calculated, and sent back to the host simultaneously. This way a better utilization of the bus is achieved.*

The second implementation used a method of software pipelining, i.e. handle the first batch in a special way, followed by a loop to handle asynchronous receiving, processing and sending the data, and an epilog.

#### A general overview of the second streams implementation

cublasSetMatrixAsync**(**d\_A **+** 0**\***size\_A**);**

cublasSetMatrixAsync**(**d\_b **+** 0**\***size\_B**);**

**for(**int i **=** 0**;** i **<** **(**iteration\_num **-** 1**);** i**++)** **{**

cublasSetMatrixAsync**(**h\_A **+** **(**i **+** 1**)\***size\_A**);**

cublasSetMatrixAsync**(**h\_B **+** **(**i **+** 1**)\***size\_B**);**

cublasSetStream**(**streams**[**i**%**streams\_num**);**

cublasSgemm**;**

cublasGetMatrixAsync**(**d\_C **+** **(**i**%**streams\_num**)\***size\_C**))**

**}**

cublasSetStream**(**streams**[(**iteration\_num **-** 1**)%**streams\_num**);**

cublasSgemm**;**

cublasGetMatrixAsync**(**d\_C **+** **((**iteration\_num **-** 1**)%**streams\_num**)\***size\_C**);**

The two methods had shown improvement over the use of the regular cuBLAS library. But more importantly, enabled us to multiply even greater matrices.

## Utilizing the CPU and GPU

### Using cuBLAS-XT

The next challenge was to perform calculations on multiple GPUs at once. For the experimental part, 2 Nvidia graphics card were used:

* Nvidia GT 430, with 96 CUDA cores, 2 SMs and a compute capability of 2.1. [8][6]
* Nvidia GT 520, with 48 CUDA cores, 1 SM and compute capability of 2.1. [9][6]
* We have used CUDA 7.0 [10] API throughout the project.

The new Nvidia cuBLAS-XT library, available starting CUDA 6.5 makes it possible to do calculations on multiple GPUs with a lot less effort. The functions built into the library, allows to choose the number of GPU’s desired for the calculation, and most of the work is performed by the library functions. The library also allows to distribute workloads between CPU and GPU.

Very little documentation currently exists for this new library, and many of the functionalities were figured out during the project.

### Using the CPU

CuBLAS-XT enables to define a workload distribution parameter. This parameter, given as a fraction, would be calculated on the CPU whilst the rest of the data would be calculated on the GPU(s). The functions was integrated as part of the mex files. CuBLAS-XT sgemm function expects to work with standard BLAS functions format usually follow this pattern:

SGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,

B, LDB, BETA, C, LDC)

See reference for further explanation [11]

A lot of work was done to enable the integration of third-party BLAS libraries with cuBLAS-XT. cuBLAS-XT methods expect a certain format of the host function. Trial and error led to the final result where the CPU and GPU methods combined results give a correct calculation.

There are several libraries which provide BLAS functions for regular CPU calculations. We have tested the built-in BLAS library of MATLAB as well as an open source implementation open-BLAS[12]:



Figure 3: Open BLAS vs MATALB BLAS

*This figure shows a slight advantage for MATALB BLAS routine over the open BLAS implementation.*

Eventually MATLAB BLAS libraries were chosen for their slightly better performance. Using MATLAB’s built in BLAS library also allows for a simpler integration, as no additional external libraries are needed for our function to work.

A comparison was made between CUBLAS-XT and our attempts to implement the method using streams. For reference, a comparison to CPU calculation time is also shown.



Figure 4: Streams compared to CuBLAS-XT

*The figure shows an advantage to using CUBLAS-XT over similarly performing methods using streams.*

## Using CPU GPU and FPGA

Using this library, the workload is split twice. First, the data is split between the GPU devices and a host device.   
Given the CPU usage wanted is and matrices

The CPU handles the calculation of

.

The GPU then handles the rest of the calculation, i.e  
 .  
Second, the host would handle distributing the workload between itself, and the FPGAs in the server. Once all components have finished their calculations, the result of each hardware component is returned, and summed up for the final result.

### Using the FPGA

Several other teams were working on integrating the FPGA in the workflow for the final calculation. Our task was to provide the infrastructure to enable FPGA integration with MATALB.

To achieve this, pthread library[13] was used to split the workload across the devices, where the CPU acts as the control center. The idea is to allocate a separate thread for every FPGA card available. This thread would handle transferring data to and from the device.

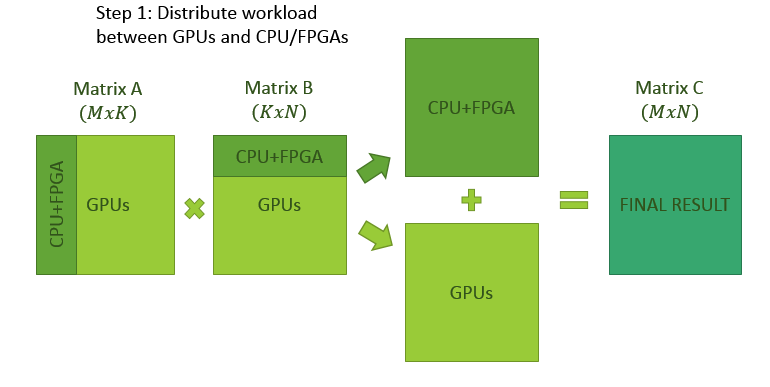


Figure 5: Step 1, divide workload between CPU/GPU

*The figure demonstrates the way the workload is distributed between the host and the devices.*

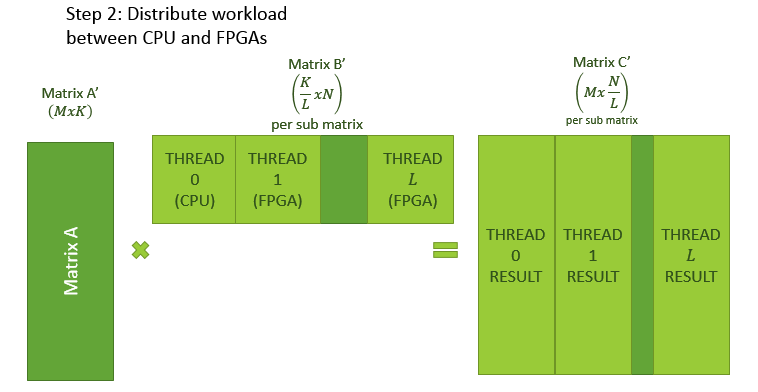


Figure 6: step 2 divide workload between FPGA and CPU

*Here the distribution between the CPU and FPGA’s is shown. Assuming there are L FPGA devices available, L threads would handle data transfer to the devices, and an additional thread to handle CPU calculation.*

Currently only a simulation of this complete workflow has been achieved, where the CPU is doing both the distribution of the workload and the calculations themselves. This is because the FPGA was not ready to perform actual calculations. Multicore CPU’s are best used for this workflow, as they make it possible handle data transfer and calculations independently between the cores.

Currently the infrastructure for FPGA integration is commented out in the final function. Once a working function is implemented, we expect an identical method signature to the SGEMM routine currently in use. [11]. This routine would then replace the existing sgemm routine in run\_fpga\_blas.

MATLAB is creating the matrices in a column major format, converting between the formats might be a costly process. The following example explains what the leading dimension of the matrix must be for correct usage of the function. This is critical, as we pass the data in **column major** format when dividing the workload across the FPGA cards. This needs to be considered when creating the FPGA function itself or a wrapper for the function.

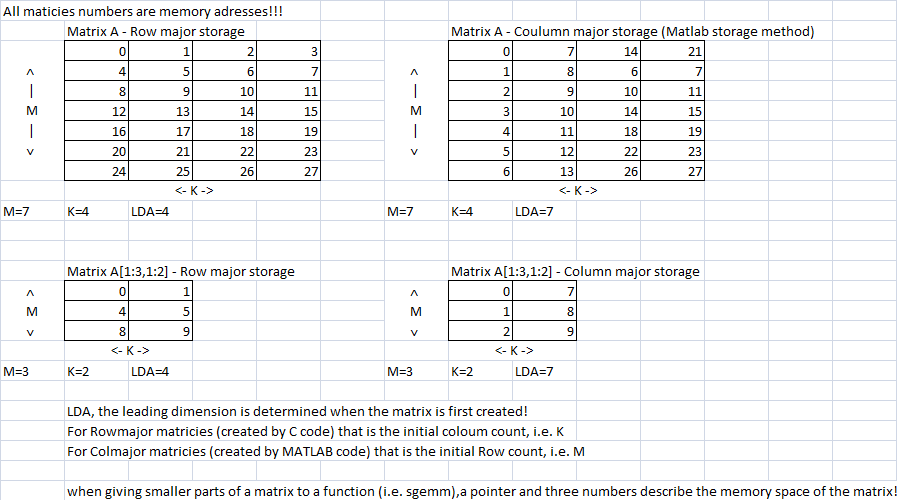


Figure 7: Leading dimension explained

### Final integration

Lastly, a MATLAB wrapper function was created to make for a simple MATALB user interface. The function enables to choose the desired parameters

MatrixMulEnhanced( A , B , thread\_num , GPU\_num , CPU\_usage , FPGA\_usage , xt\_blk\_dim )

* A - Left side matrix, must be single type NOT double.
* B - Right side matrix, must be single type NOT double.
* thread\_num - Number of threads to use, 1 is default if CPU percentage is entered. If you wish to use the fpgas on system, enter (SYSTEM\_FPGA\_NUMBER + 1) as this parameter.
* GPU\_num - number of on system GPUs to use.
* CPU\_usage - percentage load of CPU to use while performing the operation.
* FPGA\_usage - percentage load of fpgas to use while performing the operation.
* xt\_blk\_dim - block dimensions for BLAS\_xt routine, for more info, check CUDA documentation.

See chapter ‎9 and code documentation for complete explanation on using/compiling the method.

# Results

To show the method is showing real speedup, it was first compared to standard MATLAB matrix multiplication.



Figure 8: CuBLAS-XT CPU/GPU compared to MATLAB

*This figure shows almost no difference when using the standard MATLAB multiplication when comparing it to CUBLAS-XT, when only using the CPU. When only the GPU is used, there is a 2x speedup for CUBLAS-XT (for the largest matrix), as expected.*

Knowing there is a clear advantage to using the GPU, we expected even better results when combining the calculation power of both CPU and GPU. Our method, combined with CUBLAS-XT allows to define a percentage for workload distribution across host and device. We expected to find a minima point for a given matrix size.



Figure 9: Varying CPU usage percentage

*This figure shows a minima at around 40%. This means 40% of the workload on the CPU, and the rest was performed on the devices and combined for a final result. This test used 2 GPU’s GT 430 and GT 520.*

The results match our expectations, meaning the combined calculation times are significantly smaller than just using the CPU (100%) or just using the GPU (0%).

The speedup has a lot to do with the way cuBLAS-XT handles the data transfer and calculation in parallel. This could be seen in this image.



Figure 10:Nsight of final version

*This image shows the device is busy during most of the time and does not wait for all the data to be transferred first. The use of streams is also apparent without them being explicitly defined by the user.*

The next observation is that what we actually got are 2 somewhat linear lines that have a point of connection. This can be explained when we look at the simple math behind the calculation, when using the formula from ‎5.2 and take into account that we used only a single dimension with the test:

This is an approximation of the calculation time, since the constants and time constants vary slightly form test to test

What we see is the falling line that starts on the left is the GPU run times, since it is getting the most element therefore it is the maximum number between the two expressions, and after the cutoff point the rising line is the CPU calculation times, this is further confirmed by noticing that 0% point (GPU only) is lower than 100% (CPU only).

After understanding this graph we thought that we could find the optimal point by running only 4 tests since that would define the 2 linear lines, however, after many test we saw that the error we got was far too high, and the lines are only mostly linear and change depending on the matrices used in the test.

We have tried to come up with the best way to find the optimal CPU usage percent. Our test have showed that this point might vary a lot, depending on the hardware in use, the size of the matrices multiplied, etc. We compromised on finding the optimal calculation point by trying the calculation with different ratios, and using the one which gave the best results. Assuming there is a global minima, we used binary search to find the optimal point. This method performs few calculations and gives decent results. To really find the optimal point for a given hardware and workload, it is best to run the full test. If a certain server is to perform many calculations on similarly sized matrices, it would pay off to determine the optimal settings once, and use them for future calculations.

In general, for our hardware around 40% CPU load was optimal point for the largest matrices the hardware could handle.

# Concluding remarks

We improved on MATLABs ability to perform large matrix multiplication by combining the computational resources of the CPU and GPU. This was the main goal of the project. It is reasonable to believe adding the FPGA would further improve the performance, although finding the optimal operation point for such a configuration might be even more difficult and time consuming. The final result could be easily integrated and used by any user with an Nvidia card and MATLAB.

# Table or meanings

* SM: Streaming multiprocessor
* Compute capability: The technology used on a given Nvidia card. Higher computing capability enables more features [6].
* Host: The PC running the function
* Device: GPU’s and or FPGA connected to the host.

# Code documentation

## Directories

* Final version: Contains all the files and directories needed to run and/or compile the functions in a MATLAB environment. Will be referred to from now as “the main directory”.
* FPGA: Contains the libraries needed to integrate the existing FPGA driver to the code. FPGA integration is only possible with Virtual-Studio 2013 installed.
* Pthreads\_Win32: The directory contains the pthread library needed to install the function with threads. Pthreads are used to distribute data transfer from CPU to FPGA’s. This is an open source library containing all the needed code for it to run [13]
* VS 2010 xml: Running the function requires the correct .xml file to be in the working directory where the function is executed. The directory contains the .xml file needed to integrate the function with Visual Studio 2010 edition.
* VS 2013 xml: Same as the previous, but for integration with VS2013 edition.

## Files

* MatrixMulEnhaced.m: This is the wrapper MATLAB file for our function. The top of the file contains the signature of the function and a detailed explanation on the different parameters the function could receive.
* MatrixMulMex\_1\_0.cu: This is the main file containing most of the code. In-depth documentation of the functions later in this file.
* MatrixMulMex\_1\_0.mexw64: The compiled file version. This file is created once the compilation process is successful. Having this file and running the function in the final\_version working directory is enough to have a working function. No need to compile.
* matrixMulMex\_1\_0\_open\_blas.cu: This is the previous version of the file using the open\_blas library [12]. As stated before, this option was tested but not selected due to better performance of the built in MATLAB library.
* MatrixMulMex\_tests.m: The file sets up the environment for the function and contains examples and test for the function.
  + First there are detailed explanations on how to compile and install the function in a MATLAB environment.
  + Environment Set UP: For the function to run and compile, directories and libraries in the main directory need to be added to the system PATH and linked for compilation. To make the experience easier for the user, all of this is done by running the Set-Up from inside MATLAB. Cuda installation folders need to be updated to the actual installation folders on the computer. (Currently the value is the default installation path).
  + Function Usage Reference: An example for using the function. Creates 2 matrices and multiplies them.
  + Permutes definitions and initialization for the tests:
    - Matrix size
    - Granularity of the test (Ineration\_num)
    - Number of samples per test iteration (Samples)
  + Complete test: Run on most possibilities for CPU/GPU usage and produces a graph of calculation time. This is the full test to find the optimal operation point. This might take a long time, depending on the matrix size, granularity and number of samples.
  + Only CPU vs only GPU test
  + Binary minimum search: Finds the optimal CPU usage percent by performing a minimum number of calculations. This test is fast but with higher error rate. Recommended to run several times if possible to get a consistent result.
* Step\_by\_step.docx: Detailed installation documentation and use case examples
* Nsight\_step\_by\_step.docx: Guide on installing and using Nsight to provide insight to how the function works.
* Final presentation: Summary of the project
* Matrix multiplication on multiple GPU and FPGA server: This document

### Main file MatrixMulMex\_1\_0.cu

void mexFunction**(** int nlhs**,** mxArray **\***plhs**[],**

int nrhs**,** const mxArray **\***prhs**[])**

This is the main gateway function between MATLAB and the C/CUDA program. Here the value of parameters is determined by using a switch case mechanism. If a parameter is not used, a default value is assigned.

int matrixMultiply\_wrapper**(**float**\*** h\_A**,** float**\*** h\_B**,** float**\*** h\_C**,** int devID**,** sMatrixSize **&**matrix\_size**,** int gpu\_num**,** int block\_dim**)**

A wrapper for cublas-XT routine. Sets up the handles for the cublas-XT routine, as well as the workload division. Executes the cublas-XT kernel.

cublasXtSgemm

A function provided by Nvidia to handle and optimize calculations of matrix multiplications across CPU and multiple GPUs. This is where the magic happens.

void cblas\_sgemm\_wrapper**(**char **\***transa**,** char **\***transb**,** int **\***m\_p**,** int **\***n\_p**,** int **\***k\_p**,** float **\***alpha\_p**,** float **\***h\_A**,** int **\***lda\_p**,** float **\***h\_B**,** int **\***ldb\_p**,** float **\***beta\_p**,** float **\***h\_C**,** int **\***ldc\_p**)** **{**

This routine is called by cublasXtSgemm. This routine handles CPU calculations and FPGA workload. Note that the routine has a standard level 3 BLAS signature. Here a thread is assign for every available FPGA and a thread for the CPU. For each thread, the inputs and output address pointers are preassigned. Note that in case the workload cannot be evenly distributed, the residue is always assign to the CPU.

void**\*** run\_cpu\_blas**(**void **\***threadarg**)**

This routine is called by the CPU thread. It invokes the built in MATLAB BLAS routine sgemm.

void**\*** run\_fpga\_blas**(**void **\***threadarg**)**

This routine is called by the FPGA threads. Currently this routine is identical to run\_cpu\_blas. In order to successfully integrate the FPGA, the existing sgemm routine needs to be replace by a FPGA routine with the same signature. Note that at the begging of this routine, the code used to integrate the FPGA driver was commented out. Using this code, the FPGA driver can be compiled and used with the function. (This code sent data to the FPGA and got the same data back and did not perform any calculation).

# Installation process

This is a short summary of the steps required to use and/or compile the function. Detailed example with step by step instructions is found at Step\_by\_step.doc.

## Regular use

* Unzip the compiled final version to a local directory.
* Add the directory to MATLAB path (or run the function from the unzipped directory)
* Use matrixMulEnhanced. See function signature for help and documentation.

## Install to compile

* Only works on 64-bit window versions (tested only on windows 7).
* Install visual studio (2010/2013)
* Install CUDA
* Install cuBLAS-XT premier license
* Unzip the final version in a local directory
* Copy the right .xml file from the one appropriate directory (2010/2013) to the main function directory
* Open MATALB and go to the main function directory. Open the file matrixMulMex-test.m
* Update the CUDA installation directory.
* Uncomments the VS2013 configuration lines if VS2013 is installed
* Run the setup inside MATLAB
* Compile using the provided compilation line in the test file
* See additional help in the README and tests file.

## FPGA integration

Following these steps, makes it possible to use the function on any pc that meets the requirements. More steps are required to enable FPGA integration into our code. Currently, the CPU simulates the calculation done by the FPGA. Integrating a working FPGA function needs to happed in the function:  
void**\*** run\_fpga\_blas**(**void **\***threadarg**)** **{**

// DWORD BUF\_SIZE = 536870912;

// UCHAR\* baseWriteBuffer = new UCHAR[BUF\_SIZE];

// UCHAR\* baseReadBuffer = new UCHAR[BUF\_SIZE];

// DMATransfer(baseWriteBuffer, BUF\_SIZE, baseReadBuffer, BUF\_SIZE);

// delete[] baseWriteBuffer;

// delete[] baseReadBuffer;

thread\_data\_t **\***d **=** **(**thread\_data\_t**\*)**threadarg**;**

ptrdiff\_t **\***m **=** **&(**d**->**M**);**

ptrdiff\_t **\***n **=** **&(**d**->**N**);**

ptrdiff\_t **\***k **=** **&(**d**->**K**);**

ptrdiff\_t **\***LDA **=** **&(**d**->**lda**);**

ptrdiff\_t **\***LDB **=** **&(**d**->**ldb**);**

ptrdiff\_t **\***LDC **=** **&(**d**->**ldc**);**

sgemm**(**

d**->**TransA**,** // Pointer to the letter T/N (transpose or not)

d**->**TransB**,** // Pointer to the letter T/N (transpose or not)

m**,** // Number of rows in left matrix

n**,** // Number of columns in left matrix

k**,** // Number of columns in right matrix

**&(**d**->**alpha**),** // Parameter. Set to 1.

d**->**A**,** // Pointer to matrix a

LDA**,** // Leading dimension of matrix A

d**->**B**,** // Pointer to matrix B

LDB**,** // Leading dimension of matrix B

**&(**d**->**beta**),** // Parameter set to 0

d**->**C**,** Pointer to matrix C

LDC**);** // Leading dimension of matrix C

**return** **NULL;**

**}**

For the FPGA to work, a driver for the FPGA must first be installed. We worked with a driver provided to us by Noam Eliyahu [noamel@campus.technion.ac.il](mailto:noamel@campus.technion.ac.il) .

We tested the function a SUME netFPGA connected via PCIe. The commented out code sends a memory buffer to the FPGA and receives it back as-is. This is of course not the effect we want. In order for the FPGA to work, the provided function needs to follow standard level 3 BLAS format. This could be done with a wrapper function, similar to the sgemm function in the current code. The wrapper function could arrange the parameters received by run\_fpga\_blas to fit the datatypes and structures used by the FPGA.

## Using Nsight

Using Nsight could be used to profile the different workloads and how the GPU–CPU array is used to calculate them. Using Nsight with MATLAB is explained in a separate guide Nsigh\_step\_by\_step.

# Bibliography

[1] “Basic\_Linear\_Algebra\_Subprograms @ en.wikipedia.org.” .

[2] “Build MEX-function from C/C++ or Fortran source code - MATLAB mex.” [Online]. Available: http://www.mathworks.com/help/matlab/ref/mex.html. [Accessed: 29-Aug-2015].

[3] “Create C Source MEX-File - MATLAB & Simulink.” [Online]. Available: http://www.mathworks.com/help/matlab/matlab\_external/standalone-example.html. [Accessed: 29-Aug-2015].

[4] “CUDA Toolkit Documentation.” [Online]. Available: http://docs.nvidia.com/cuda/#axzz3kCDPwekz. [Accessed: 29-Aug-2015].

[5] “Run MEX-Functions Containing CUDA Code - MATLAB & Simulink.” [Online]. Available: http://www.mathworks.com/help/distcomp/run-mex-functions-containing-cuda-code.html. [Accessed: 29-Aug-2015].

[6] “CUDA GPUs.” [Online]. Available: https://developer.nvidia.com/cuda-gpus. [Accessed: 30-May-2015].

[7] “NVIDIA Nsight | NVIDIA.” [Online]. Available: http://www.nvidia.com/object/nsight.html. [Accessed: 10-Oct-2015].

[8] “GeForce GT 430 | Specifications | GeForce.” [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gt-430/specifications. [Accessed: 30-May-2015].

[9] “GeForce GT 520.” [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gt-520/specifications. [Accessed: 30-May-2015].

[10] “CUDA 7 Release Candidate Feature Overview: C++11, New Libraries, and More | Parallel Forall.” [Online]. Available: http://devblogs.nvidia.com/parallelforall/cuda-7-release-candidate-feature-overview/. [Accessed: 20-Oct-2015].

[11] “lapack-blas/sgemm.html.” [Online]. Available: http://www.math.utah.edu/software/lapack/lapack-blas/sgemm.html. [Accessed: 09-Oct-2015].

[12] “OpenBLAS : An optimized BLAS library.” [Online]. Available: http://www.openblas.net/. [Accessed: 18-Oct-2015].

[13] “POSIX Threads (pthreads) for Win32.” [Online]. Available: https://www.sourceware.org/pthreads-win32/. [Accessed: 10-Oct-2015].