Multiplication of two matrices
Start with square matrices of size n
where n is a power of 2

P1 - Version 1 with cuda technology:

Use one thread to compute each element of the solution matrix. To maximize the size of the matrices used in the multiplication, you must tile the matrix, i.e. setup blocks on the grid. The code on the last page of lecture notes – Ch4-Multiply in Cuda version 1. You should make your code as efficient as possible **without** using shared memory.

Optional: There are two software packages should be installed – Nsight Eclipse Edition and Nvidia Visual Profiler (nvvp). To use nvvp, I don't think you can use I/O redirection so you must use command line arguments to run a program. Your choice if you want to use Nsight but the profiler will be useful to you moving forward. To start the profiler use the nvvp command at the CLI or find it in the menus.

Make use of command line arguments argv and argc. To automate the process of testing a variety of tile widths with a variety of matrix sizes, you must setup the executable code to accommodate the following command line arguments.

cuda_mult_v1 <tile width> <matrix A file> <matrix B file> <matrix A*B file>

cuda_mult_v1 – name of executable

tile width – 4, 8, 16, 32 (use square tiles – why is 32 the max?)

matrix file format– contains the size (N) of a square matrix and one N x N matrix. We will only consider values of n that are powers of 2. Initially, the possible values are 2^9 through 2^{14}. Side note: If you test value of N from 2^2 through 2^4 be ready for CUDA errors: invalid configuration.

Layout

1^{St} 4 bytes of the file – N – for a NxN matrix

Each line following will contain random numbers between 0.00 and 99.00. I will use a %6.2f format which will provide at least one space between all numbers. If you use the same format in the output you can use the diff command to check your answer.

Suggestions – you need data to test your code:

Small ones you may be able to create by hand but you need to start at N = 2^5. I prefer to write a program to generate the input matrices. Labeled them **input-*width*** where *width* is the width of the matrix. Matrix A contains random numbers maybe something like random numbers between -5 and 5. Make the B matrix the identity matrix. At the same time, create the solution matrix which is a file that contains just matrix A. If you do A*B the answer should be A. NOTE: This is only a preliminary check and may hide other bugs that may not surface with multiplying by the identity. Here are my file names for a 256X256 matrix mult.

InputA-256 – contains random numbers between 0.00 and 99.00.
inputB-256 – contains the identity matrix.
output-ans-256 (this is just a copy of inputA-256 initially for testing)

See file format given above. If you do not match this format I will not be able to test your program for correctness.

Optional suggestion, write a separate program that will convert the raw data file into something you can read incase you need the values in a form that is readable. This will also be useful when you are checking the results of a mult. The diff command will be used to verify that your output from your mult program matches the answer. When you run your mult program, have it output the solution in a file that is labeled with the tile-width and matrix size. Something like ans-32-256 which represents the solution from running the program on tile width of 32 and a matrix size of 256. Please include basic error checking to make sure there are no issues with actually performing the mult requested.

At this point, you can check your solution by using the diff command as in

***diff* output-ans-256 ans-32-256**

If there are any differences, it should show up at this point. NOTE: This is not an absolute way to detect errors in your mult program. You should test it with a completely random A and B.

The general idea is to run the program several times with varying matrix sizes and tile widths. It is easier to automate the running process. One suggestion to automate the running of the program, create a bash script. Here is one version of a bash script that runs the program and generates a series of solutions. I named the script cudaRun. Make sure you set the permission to include execute for the owner. All of the input data needs to be established prior to running the script. You can tweak the script to use the **diff** command for a pseudo-check of correctness.

```
#!/bin/bash
#
# The mult_v1  program takes the name provided
#       and adds -device-tileWidth-width
#       to form the name of the answer file.
#       Example: If the program is named mult_v1
#       an executed with
#            mult_v1 32 input-A-1024 input-B-1024 ans
#       uses tile width = 32, matrix width=1024
#       The output file name argument of “ans” the mult_v1
#       creates an output file named
#          ans-32-1024
#
#       Also note that mult_v1 can output std_out the times
#       which are captured in avgTime and echo'ed to the display.
#
if [ "$#" -eq 1 ]
then
  device=$1
  run="1"
  for width in 512 1024 2048 4096 8192 16384
  do
    for tile_width in 4 8 16 32
    do
      avgTime=`mult_v1 $tile_width input-A-$width input-B-$width ans`
      numCells=`expr $width \* $width`
      echo $run $width $tile_width $numCells $avgTime
    done
    run=`expr $run + 1`
  done
else
  echo "Usage: cudaRun <device number>"
fi
```

General layout of main function.

```
int main (int argc, char *argv[])
{
   check arguments;
   Load matrix A
   Load matrix B
   Call a function that performs any cuda setup and kernel call.
   Save solution matrix
}
```

Here is the big question, what setup yields the best performance?
Report the results in two different ways.

a) Report three times – please just use gettimeofday for all times.
   Time all of the items in the main function.
   Time all of the items in the inside the cuda setup function (including the kernel call).
   Time just the call to the cuda kernel performing the matrix multiply.

b) some type of visual representation of the time data using the different sizes. Your choice.