GPU Farm 1.0
Maria Chiara Cecconi
My GitHub directory: https://github.com/MCC04/GPUfarm

Major updates:

1. project splitting in different folders
2. code splitting in different sources/headers
3. Makefile added, it contains all possible targets to compile all possible execs
4. three new Kernels have been added
5. host version, both sequential and parallel, for Cosine and Matrix Multiplication
6. new test scripts, updated datasets and inverted loops
7. Python plot script splitting in Device and Host version

1. PROJECT FOLDERS

- **bin**: contains all executables and objects that Makefile will produce;
- **images**: contains 10 sample images (useful to test image Processing Kernels);
- **include**: headers;
- **log**: holds *top* and *nvidia-smi* commands shots (taken every 10 seconds);
- **plots**: holds Python scripts that generate average measures and speedUps (all saved in CSV format to "output" subfolder), output graph images;
- **profiling**: contains the output from the *nvprof* tool, to be used to compare with Event measures taken inside the code;
- **results**: holds .txt files generated directly by the execution of code;
- **src**: source files (.cpp, .cu);
- **tests**: all bash scripts to compile, run all possible project applications and output results to mentioned above .txt files.

2. CODE STRUCTURE

- **cudaUtils.cpp**: gives some functions calling CUDA API, i.e. check for CUDA runtime API results/errors, stream creation/destruction;
- **farmkernels.cu**: contains kernels for Cosine computation and functions to set input to kernels, call kernels and return results to the caller (*main_cos.cpp*);
- **main_cos.cpp**: using c++ directives, chooses between three async modes: Future async, Stream without unified memory, Stream with UM (referred in code as "managed");
- **imageMatKernels.cu**: contains a kernel for simple Matrix Multiplication, two kernels for image blurring with both Box Filter algorithm and Gaussian blurring (this last one is still in testing phase) and clearly we have relative functions to set input, call kernels and return results to the caller (*main_imageMat.cpp*);
- **main_imageMat.cpp**: using c++ directives, chooses between: Mat Mul with one-block Unified Mem allocation for all *Nmat* matrices (suppose to have C = A * B, we have one block for all As, one for all Bs and one for Cs), Small Mat Mul allocates all matrices in a separate way (copying them between host and device memory one by one, more precisely 3 mats at time -A,B,C-), Box Blurring, Gaussian Blurring (both of them runs on 10 sample images;

- **lodepng.cpp**: it's an open source library that gives some simple functions e.g. read a .png image and return a char buffer, output a char buffer as .png image;
- **hostfarm.cpp**: using directives makes possible to choose between an host side implementation of: sequential Cosine, parallel Cos, sequential Mat Mul, parallel Mat Mul.

3. MAKEFILE

Compile <u>device</u> code with following targets:

- **future** (futurelow): compiles the Cos Kernel, with future async;
- **managed** (managedlow): compiles the Cos Kernel, with streams and Unified Memory;
- **stream** (streamlow): compiles the Cos Kernel, with streams and MemCpy;
- **empty** (emptylow): used to measure how long it takes to launch an Empty Kernel (still testing);
- **matmul** (matmullow): compiles Mat Mul Kernel on "*Nmats*" matrices all allocated in GPU mem as big single block (UM);
- **smallmatmul** (smallmatmullow): compiles Mat Mul Kernel on "*Nmats*" matrices allocated and then copied in GPU mem one at time (MemCpy);
- **blurbox** (blurboxlow): compiles Blur Box Kernel on 10 .png photos, converted to char array by "lodepng" and allocated on UM one at time;
- **blurgauss** (blurgausslow): compiles Blur Gaussian Kernel on 10 .png photos, converted to char array by "lodepng" and allocated on UM one at time (still testing).

Compile <u>host</u> code with following targets:

- **hcosseq**: Sequential Cos computation;
- **hcospar**: Parallel Cos computation;
- **hmmseq**: Sequential Mat Mul computation;
- **hmmpar**: Parallel Mat Mul computation;

Typing "**make clean**" the folder /bin will be emptyed.

Targets in parentheses (marked as *low) are similar to the original version but they use a small amount of parallelism:

- *GridDimensions*=(1, 1, 1) and *BlockDim*=(16, 1, 1) in Cos/Blur Box;
- *BlockDim*=(4, 4, 1) in MatMul.

While High parallelism is:

- *GridDimensions*=(N/BlockDim, 1, 1) and *BlockDim*=(512, 1, 1) in Cos/Blur Box;
- *GridDim*=(M/block.x, N/block.y, 1) and *BlockDim*=(32, 32, 1) in MatMul.

4. NEW KERNELS

1. `matMulKernel(float* Ad, float* Bd, float* Cd, int m, int k, int n)`
   computes simple matrix multiplication between A[MxK] and B[KxN] giving C[MxN]. Here we have a for loop [*0, (k-1)*];

2. `void blurBoxFilterKer(unsigned char* input_image, unsigned char* output_image, int width, int height)`
   simple blurring, for each channel (RGB) of each pixel we sum an "average" of adjacent pixels (the filter size was set to 5). Here we have two nested loop, each running in [*-filterSize, +filterSize*];

3. `gaussianBlurKer(const unsigned char* const inputChannel, unsigned char* outputChannel, int numRows, int numCols, const float* filter, const int filterWidth)`
   an helper function computes the Gaussian matrix (the filter), then I pass it to the Kernel.
   Here, for each pixel of the input image, we center the filter with that. To compute value of blurred pixel, we multiply aligned pixels, then we sum up all obtained multiplication results. This last result will be the blurred pixel.
   We have two for loops inside the Kernel, each running in [*0, filterWidth]* - the filter width was set to 5.

5. HOST COS AND HOST MAT MUL

Sequential version of Cos an MM are trivial.
Parallel versions are implemented exploting **FastFlow** framework:

- **Host Cos**: I'm using *ff_farm* skeleton where I've implemented Emitter as *ff_node*, giving N random values to the output channel. The Worker gets as input the random values generated by the Emitter. The worker is an *ff_node* that applies Cos on the input value for M times.
- **Host MM**: Again I'm using *ff_farm* skeleton. We have *Nmats* matrices allocated in a single block for A, and the same for B and C.
  The emitter send in out channel the taskId (it's from 1 to *Nmats*), this is used by the Worker (*ff_node*) to compute the index of the corresponding matrix in the block (holds for A, B and C) to be picked, then Worker computes MM on A, B and C.

6. NEW TEST SCRIPTS

As discussed in the last meeting, I had some spikes in Stream and Future measures. So it was needed to introduce two type of execution tests for each of the executables to run:

- first type is running for N times all possible datasets;
- second type is running each dataset for N times (marked as *_invert in the script names).

I used N=13, to remove outlayers (for the moment I only delete the max and the min) and then compute the average on the remaining values.

The most important tests are:

- **run_devcos.sh**: runs device Cos test and its inverted version;
- **run_devmat.sh**: runs device Mat Mul (and image proc) test and its inverted version;
- **run_lowpar.sh**: runs all device kernels with reduced Block and Grid dimensions;
- **run_hostcos.sh**: runs host Cos test and its inverted version;
- **run_hostmat.sh**: runs host Mat Mul test and its inverted version;
- **perf_monitor.sh** and **host_perf_monitor.sh**: these two scripts are launced in background at the beginning of all the above scripts. They make *nvidia-smi* and *top* stamps, every 10 seconds, then put them in files inside /log folder.

The measures picked in both normal and inverted versions, seem to haven't spikes (that we supposed as due to CPU load). In fact, watching at *top* output during executions, the CPUs appeared to have a low load.

PROBLEMS

I found an anomalous behavior, in both device and host versions.
In particular, for device Low Par version is faster than High Par. Similar for Host, sequential is faster than Parallel.
For the latter case, I've already an idea about the reasons. Basically I think my parallel implementation isn't good and I've already think on ways to improve that. I'd like to discuss about that in the next meeting.
The former case, sounds really strange for me. I'm searching and reading from docs and forums, but at the same time I'm debugging my code for possible hidden errors.

Here I mention some examples of measures about device low/high par problem:
(measures refer to GPU event times in millisec)

FUTURE EVENT LOW PAR					
M	**10**	**50**	**250**	**1250**	**2500**
N					
3584	0.134949	0.140832	0.197623	0.459182	0.784840
7168	0.260410	0.289518	0.383165	0.911956	1.586233
14336	0.608794	0.643444	0.862245	1.905639	3.212412
28672	1.391863	1.487284	1.927258	4.022020	6.69548
57344	4.332476	4.54011	5.376845	9.68355	14.865445

FUTURE EVENT AVG					
M					
N	**10**	**50**	**250**	**1250**	**2500**
3584	0.146336	0.162231	0.242161	0.659191	1.176816
7168	0.279618	0.314338	0.499482	1.304298	2.339808
14336	0.642810	0.696334	1.031342	2.69250	4.790432

28672	1.558993	1.653888	2.347675	5.69537	9.802737
57344	4.735022	5.355839	8.149437	21.76384	38.65062

FUTURE EVENT SPEEDUP					
M	**10**	**50**	**250**	**1250**	**2500**
N					
3584	0.922191	0.868094	0.816080	0.696584	0.666918
7168	0.931303	0.921039	0.767124	0.699193	0.677933
14336	0.947082	0.924044	0.836042	0.707758	0.670589
28672	0.892796	0.899265	0.820921	0.706190	0.683021
57344	0.914985	0.847693	0.659781	0.444937	0.384610

This behavior is completely analogous for Stream and Managed.

MAT MUL EVENT LOW PAR					
Num Mat	**8**	**16**	**32**	**64**	**128**
Mat Dimensions					
16	6.160407	9.449503	19.957836	39.410136	70.047109
32	6.149265	11.028190	21.900736	43.676836	90.5018
64	5.090694	9.2459709	18.074745	38.380245	59.918727
128	7.584183	15.969636	25.378790	51.980536	99.939881
256	8.629600	19.756218	43.011963	85.385418	171.592454

MAT MUL EVENT AVG					
Num Mat	**8**	**16**	**32**	**64**	**128**
Mat Dimensions					
16	6.193335	9.697422	19.012018	38.5110818182	72.0791636364
32	6.375108	11.576872	22.596118	45.1155363636	91.3540636364
64	5.969623	9.6451418	20.246163	40.8183545455	69.0095818182
128	8.604489	18.243336	30.963245	63.4300454545	129.4382727273
256	14.5162454	29.404154	62.91078	123.369636363	250.4001818182

MAT MUL EVENT SPEEDUP					
Num Mat	**8**	**16**	**32**	**64**	**128**
Mat Dimensions					
16	0.9946832879	0.9744345381	1.0497484366	1.0233453464	0.9718080171
32	0.964574291	0.9526053511	0.9692256071	0.9681107637	0.9906707638
64	0.8527664147	0.9586143038	0.8927491538	0.9402692951	0.8682667782
128	0.8814217272	0.8753681917	0.8196424676	0.8194939163	0.7721045693

256	0.5944788503	0.6718852654	0.6836978081	0.6921104795	0.6852728832

This behavior is completely analogous for Small MM, Lots Small MM, Blur Box.

As we can see in both of the above examples, Low parallel is a bit faster than High parallel. This gives us a non-existent “GPU speedup” .

ToDo List:

- Solve the device Low/High Parallel problem
- Change some data sets: I found out that some data sets aren't correct in order to use almost all of 56 SM in Tesla P100;
- Implement Mat Mul Kernel exploiting shared memory, faster solution;
- Implement host version for image processing;
- Optimize Parallel Host versions for Cos and Mat Mul;
- Format output in a better way (put parameters and results as one line output);
- I'm already reasoning about how my work can be “merged” with CPU/GPU mix (by T. Serban), so I'll soon give a draft about that;
- Solve a little problem in Makefile, can't understand why it recompile the exec (.out) always even if nothing has been modified. While for the other object it doesn't recompile if not needed;
- Improve visual result for Python Plots.