Application of CUDA in DSP

Tomáš Mazanec
mazanec@utia.cas.cz, +420-2-6605 2472...

Contents

Revision

Revision	Date	Author	Description of document changes
0	20.7.2009	T.M.	New document
1			
2			
3			
4			

1 Introduction

NVIDIA® CUDA™ is a general purpose parallel computing architecture that leverages the parallel compute engine in NVIDIA graphics processing units (GPUs) to solve many complex computational problems in a fraction of the time required on a CPU. It includes the CUDA Instruction Set Architecture (ISA) and the parallel compute engine in the GPU. To program to the CUDA architecture, developers can, today, use C, one of the most widely used high-level programming languages, which can then be run at great performance on a CUDA enabled processor.

Since a hardware with CUDA capability became available during early 2008, we decided to learn CUDA technology and evaluate some experiments in area of Digital Signal Processing. First HW used for experiments was CUDA v1.1 capability graphics card with G80 family GPU on board. Later experiments were realized on CUDA v1.3 capability graphics card equipped with GTX200 GPU. Development software suite (CUDA toolkit and SDK) we used, evolved from version 1.1 to 2.2 nowadays. Technical specifications of platforms used are outlined in appendixA.

The first evaluated CUDA implementation is a signal filtering with Finite Impulse Response digital filter (FIR). Next implementation is enumeration of Cross Ambiguity Function (CAF) on sampled radio-frequency signals. Note that chosen arithmetic was single-precision floating point in all cases.

2 Signal filtering with FIR filter on CUDA platform

Digital filters with finite impulse response are considered as basic application of digital signal processing. Filtering with FIR can be described in time-domain as equation for output signal $y(n)$:

$$y(n) = \sum_{k=0}^{M-1} h(k)\, x(n-k) \, , \tag{1}$$

where M is the filter order, $x(n)$ is input signal, $h(k)$ is filter impulse response and zero initial conditions are assumed where $(n-k) < 0$.

2.1 Evaluated implementations of FIR filter

Since the execution model of CUDA platform recognizes groups of threads as elementary parallel batch and our implementations follow this model, the FIR filter output enumeration is done in parallel threads context of CUDA. There are two possible ways how to implement FIR described by (1) in parallel:

1. Each thread enumerates one output sample within M-length loop, i.e.: dot product of filter response vector $h(0..M-1)$ and input vector $x(n-M+1..n)$ is done by one thread for a fixed n, while the other threads enumerate dot products for a different n-values.

2. Group of M-threads enumerates one output sample within data-length loop, i.e.: vectors of filter response $h(0..M-1)$ and of input $x(n-M+1..n)$ are multiplied element-wise by M-threads in parallel and then follows parallel sum operation to compute the output sample. Value of n is advanced in next loop iteration.

Implementation of the second approach to FIR has shown poor performance during initial experiments and thus it has been discarded from final implementation.

The first type of implementation approach satisfied in experiments and it has been extended and evaluated. There should be mentioned several considerations with implementation. Since the HW of GPU puts some limits to CUDA, namely maximal number of threads per block, the FIR implementation is also limited. Obviously the implementation can be extended to overcome these limits, but with price of lowering performance. The number of threads per block (<=512 at CUDA capability 1.3) means the largest parallel group of threads that can be computed at once. Thus for filters with order grater than this number, implementation was extended. This evaluation led to two separate implementations marked (I) and (II). The second one (II), which has limited FIR order to max. 512, allows to compare performance decrease of the first and extended implementation (I).

Suitable utilization of available GPU threads and memory accesses have to be done to achieve some performance of implementation. Implementation I use several performance optimization techniques of CUDA. Texture memory access is used for both filter coefficients and input vector. There is benefit of cached access to global memory and disadvantage with texture address limit, about 2^{27} in case of 1-dimensional array. Shared memory (on chip GPU) is used to store immediate output values. There is benefit of cache for write access to global memory, but disadvantage of limited size of shared memory. This limit is about near to 4k of 32bit floats.

2.2 FIR filter implementation results

Following tables (Tab. 1 and Tab. 2) present achieved results of both implementations (I and II) of FIR filter. Implementation II has limited filter order to max. M=512, but Implementation II is capable of greater filter order, up to limit of shared memory on GPU, where immediate results are stored (this limit is close to 4k array of 32bit floats). Comparison between early experiments on CUDA v1.1 and later with CUDA 2.x can be seen on the first table, Tab. 1.

Evaluation of utilized GPU threads and thus computing performance is presented on table Tab. 2. The table shows relation between performance and desired FIR order with a given input vector length. Graphic representation of this relation is depicted on Fig. 1.

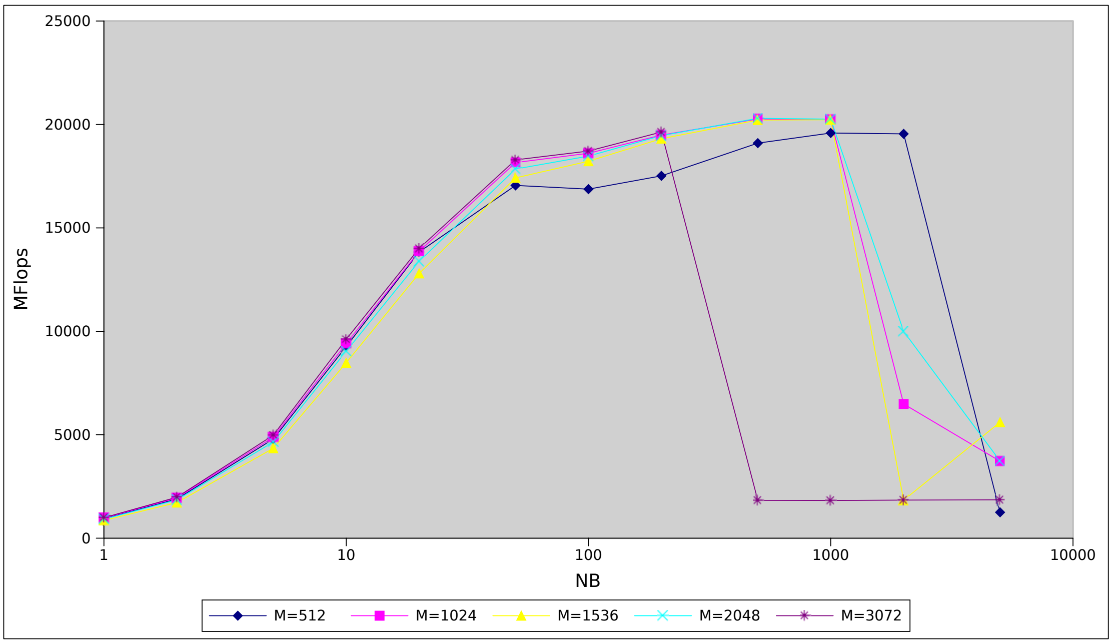

Figure 1: Graph of relation between computing performance and desired FIR order for a given input vector length (created from Table 2).

CUDA v1.1

implementation I

M	512	1024	2048	3072
N	3585	3073	2049	3073
y_len	71681	36865	18433	12289
MULs [x 1e+06]	37	38	38	38
Mflops	3250	3300	2450	2850

implementation II

M	512
N	3585
y_len	71681
MULs [x 1e+06]	37
Mflops	3050

CUDA v2.1

implementation I

M	512	1024	2048	3072
N	3585	3073	2049	3073
y_len	71681	36865	18433	12289
MULs [x 1e+06]	37	38	38	38
Mflops	13800	11200	8350	3970

implementation II

M	512
N	3585
y_len	71681
MULs [x 1e+06]	37
Mflops	14000

Table 1: FIR filter implementation results - performance in MegaFlops (M is filter order, N is length of vector segment stored in shared memory, y_len is total length of output vector and MULs denotes number of multiplying operations enumerated).

CUDA v2.1, implementation I

	Mflops					
NB	1	2	5	10	20	50
M=512	947	1893	4791	9263	13834	17054
NB	1	2	5	10	20	50
M=1024	969	1943	4879	9402	13867	18158
NB	1	2	5	10	20	50
M=1536	859	1721	4342	8478	12798	17417
NB	1	2	5	10	20	50
M=2048	917	1855	4649	9036	13386	17845
NB	1	2	5	10	20	50
M=3072	990	1979	4978	9613	14006	18289
	Mflops					
NB	100	200	500	1000	2000	5000
M=512	16874	17515	19097	19586	19546	1250
NB	100	200	500	1000	2000	5000
M=1024	18608	19488	20266	20237	6488	3721
NB	100	200	500	1000	2000	5000
M=1536	18223	19312	20212	20229	1838	5606
NB	100	200	500	1000	2000	5000
M=2048	18471	19461	20297	20258	9997	3729
NB	100	200	500	1000	2000	5000
M=3072	18715	19635	1830	1823	1841	1853

Table 2: FIR filter Implementation I - performance in MegaFlops (M is filter order and NB is enumerated number of blocks executed on GPU, which depends on input vector length).

3 Cross ambiguity function on CUDA platform

Cross Ambiguity Function (CAF) is a signal processing task necessary in Passive coherent location systems. CAF represents PSD distribution of the cross-correlation between direct and reflected signals and it's definition is:

$$\mathrm{CAF}(\tau, k) = \sum_{n=0}^{N-1} s_1(n) s_2^*(n+\tau)\, e^{-j2\pi kn/N} \tag{2}$$

where the signals $s_1(n)$ and $s_2(n)$ in time-domain are cross-correlated with τ-shift parameter and Fourier transformed. All signals/vectors are assumed to be complex numbers.

Passive coherent location needs an accurate, effective and efficient implementation of CAF. Several properties, necessary for PCL, of input signals and demands on output function have to be mentioned. Sampling rate of input signals is in range of hundreds of kilo-Hertz (e.g.:200kHz), integration period (FFT size) is large (e.g.: 128k samples) and number of enumerated delays is in hundreds of samples (e.g.: up to 600). Finally there is consideration of PCL system performance thus the time period in which a single CAF enumeration have to be finished (e.g.: 0.5sec).

3.1 Evaluated implementation of the CAF

Given application of the CAF, we implemented, demanded a Matlab interface and so the implementation use MEX-to-CUDA interface for input signals and results exchange. The implementation processes in following steps:

1. MEX: memory allocation, handle the input vectors
2. CUDA: copy data to device
3. CUDA: kernel multiplies input vectors, 600-times
4. CUFFT: 1-D 128k FFT on all 600 vectors
5. CUDA + MEX: copy data to host, handle the results

Let's note some implementation details that had been evaluated within CAF development on CUDA platform. All data are complex and single-precision (i.e: two floats per one number). Input data are two 128k vectors in this application, but the amount of data processed on GPU device is huge, in this case it is 128k*8B*600 = 600MB. Output data are only a few hundred spectral lines so it is a vector of a few hundred complex numbers (pairs of floats).

3.2 Achieved results and comparison

CAF implementation on CUDA has achieved important results which have met desired time constrains. Other goal of CAF evaluation was comparison to former implementations.

Computed CAF task had a following properties:
input signals length: 128k samples
FFT size: 128k samples
number of enumerated delays: 600
time constraint: $<=$0.5sec

Achieved computation time on earlier HW (8600GT 512MB, CUDA v1.1, CPU 2.4GHz) with our implementation with MEX interfacing CUDA routine is 2.0sec compared to Matlab routine with 3.9sec computation time. Later HW platform (GTX260 896MB, CUDA v2.1, CPU 3.2GHz) brought short computation time of both MEX interfacing CUDA routine, i.e.: 0.44sec achieved, and standalone Matlab routine, i.e.: 2.8sec achieved.

There is an interesting case to compare. Our first performance driven implementation in Q4 of the year 2006 brought results on Xilinx FPGA platforms. There was 40bit fixed point arithmetic used and customized Fourier transform engine for CAF computation. Resulting computational times were reached: 1.9sec on Virtex2 FPGA platform and 1.3sec on Virtex4 FPGA platform. More information can be found in [5] and [6].

HW	MEX & CUDA	Matlab
8600GT, CUDA v1.1	2.0sec	3.9sec
GTX260, CUDA v2.1	0.44sec	2.8sec
Virtex2 (FPGA, year 2006)	1.9sec	
Virtex4 (FPGA, year 2006)	1.3sec	

Table 3: Summary of CAF computation time results

Resulting CAF output computed on GPU with real-measured input signals is depicted on Fig. 2. More important is the relative error of CUDA implementation against Matlab standalone routine with implicit double-precision. This relative error is depicted on Fig. 3. Achieved computation times are summarized in the Table 3.

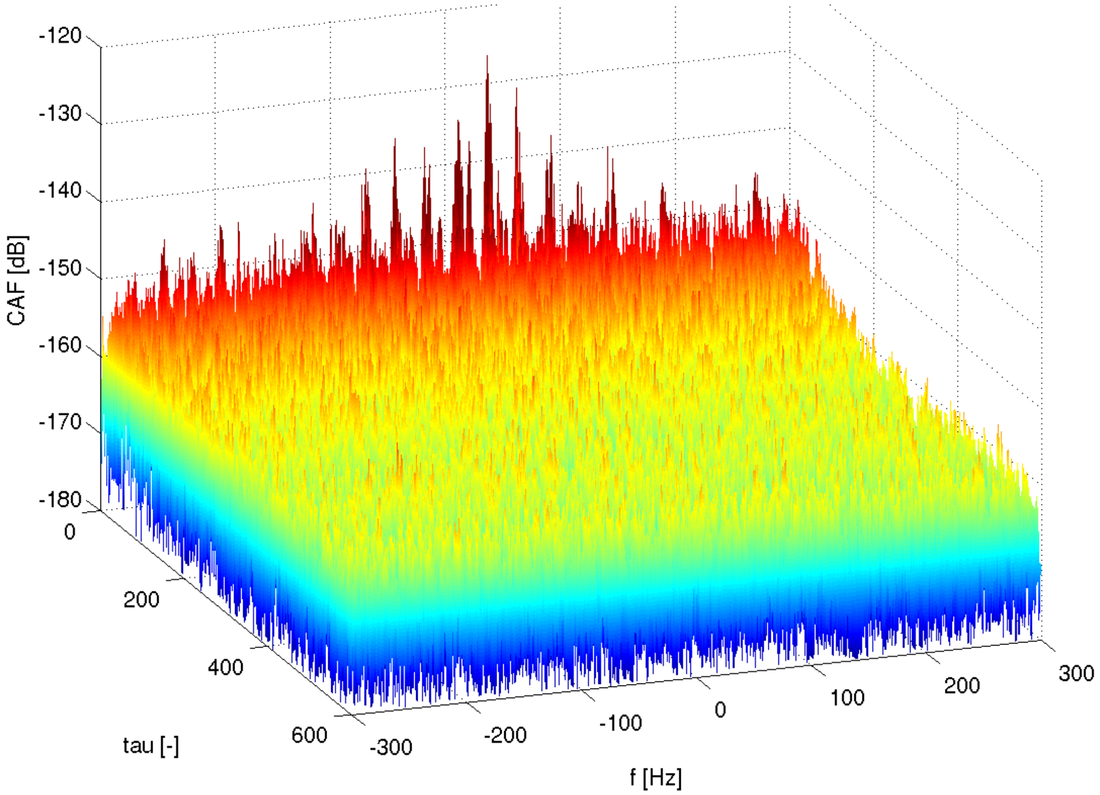

Figure 2: CAF output example for a real world radar signals.

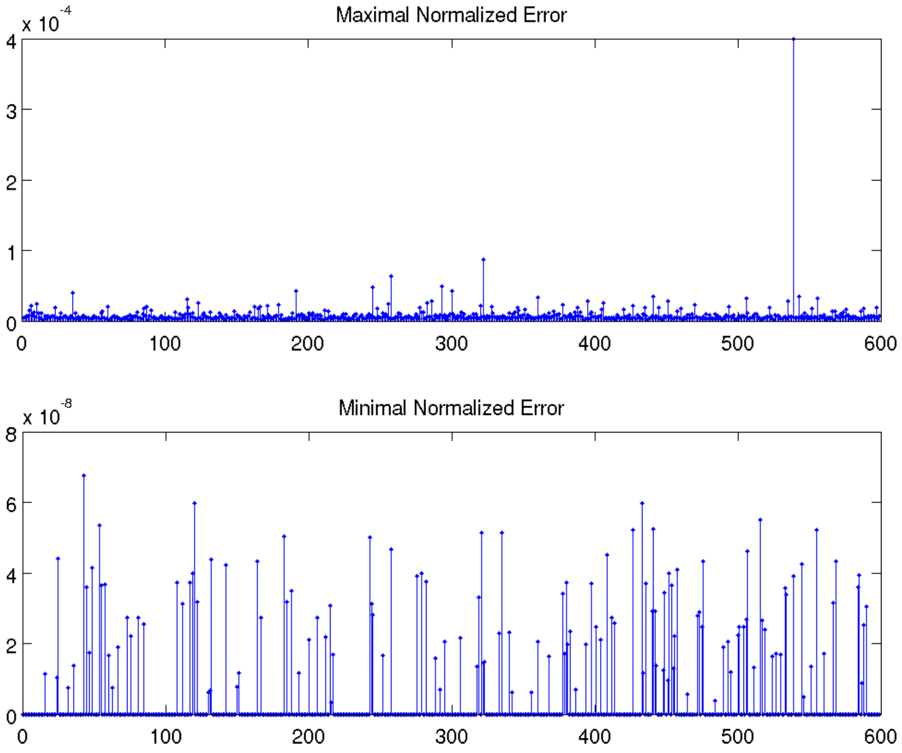

Figure 3: CAF - relative error of CUDA single-precision implementation compared to Matlab double-precision.

4 Conclusion

4.1 FIR filter implementation

Evaluation of two different approaches of implementation showed that there is always necessary to find how to fit the given algorithm to CUDA platform, CUDA thread context respectively. Winning approach (number 1 from section 2.1) was selected for next experiments.

The GPU hardware limits were taken to consideration and two other implementations were presented (marked I and II). Implementation I, the extended one, had shown that an extension of FIR algorithm lowers resulting performance for higher FIR orders, but under constant number of multiplication operations (Tab. 1).

Utilization of GPU threads have significant influence on resulting computational performance. Occupation of GPU which is leading to the performance is summarized in table Tab. 2 for a different FIR orders and input vector lengths.

Despite of all considerations that have to be taken the CUDA implementations are capable to achieve significant performance, that can reach to 20 GFlops in the case of FIR algorithm and later HW platform (within *single-precision* floating point number representation).

4.2 CAF implementation

The later CAF implementation on CUDA platform has achieved important results which have met desired time constrains (within *single-precision* floating point number representation). This time constraint is 0.5 sec to finish one CAF enumeration.

The GPU hardware limits were hitting only in the amount of necessary memory space on graphics card. The CAF implementation is capable to split the computation routine in L-loops with effort to fit the total amount of data to smaller memory space, e.g.: 8600GT/512MB graphics card.

There is an interesting case to compare. Our first performance driven implementation in Q4 of the year 2006 brought results on Xilinx FPGA platforms. There was 40bit fixed point arithmetic used and customized Fourier transform engine for CAF computation. Resulting computational times were reached: 1.9sec on Virtex2 FPGA platform and 1.3sec on Virtex4 FPGA platform. More information can be found in [5] and [6].

Finally the table Tab. 3 summarizes all the present and former achievements of our CAF implementations.

A Technical specifications of used hardware platforms

- Early implementations: GeForce 8600GT 512MB, personal computer with Intel Core Quad CPU at 2.43GHz and PCI-Express bus v1.0. CUDA toolkit and SDK version 1.1 on Ubuntu Linux v7.10.
- Recent implementations: GeForce GTX260 Core 216 896MB, personal computer with Intel Core Duo CPU at 3.16GHz and PCI-Express bus v2.0. CUDA toolkit and SDK version 2.1 on Ubuntu Linux v8.04.

B Installation and running of the implementations

Source code of all CUDA implementations should be put into ”projects” sub-folder placed in the folder where the NVidia CUDA SDK is installed, e.g.:

```
SDK_INSTALL_PATH="${HOME}/NVIDIA_CUDA_SDK"
```

Implementations of FIR and CAF are then placed in separate folders:

```
${SDK_INSTALL_PATH}/projects/tFIR_implementationI/
${SDK_INSTALL_PATH}/projects/tFIR_implementationII/
${SDK_INSTALL_PATH}/projects/tCAF-mex/
${SDK_INSTALL_PATH}/projects/tCAF/
```

Source codes of implementations are equipped with “GNU make Makefile” files, thus compilation of sources is natively targeted to linux platform. Source codes are platform independent and so can be compiled and ran on other platforms supported by CUDA SDK.

B.1 Running the FIR filter

Execution of binary is possible with or without command-line options, e.g.:

```
./tFIR_implementationI -NB=10 -M=16
```

Command-line options are:
`M` - desired FIR filter order, in range: 1..3584 for Implementation I (or up to 512 for Implementation II
`NB` - number of blocks to be enumerated as filter’s input vector, also means number of blocks executed

Command-line output of FIR Implementation II:

```
./tFIR_implementationII -NB=10 -M=16

FIR order M: 16 with NT 16 threads
No. of Blocks:10, where K:255, perBlock inp:4096 perBlock out:4081
        xlen:40816 , y_len:40801

Input file: ../../../projects/tFIR_implementationII/data/fir2DP.dat
Y_len = N-M+1 !!!
SUPERGOLD, casting to FLOAT according to device !
Output file: ../../../projects/tFIR_implementationII/data/input.dat
Output file: ../../../projects/tFIR_implementationII/data/output.dat
Output file: ../../../projects/tFIR_implementationII/data/reference.dat
Output file: ../../../projects/tFIR_implementationII/data/coeffRev.dat
Test PASSED
```

```
Processing time: 1.546000 (ms) and float MULs: 6.528160e+05
 MegaFlopsy: 422.261322
```

Command-line output of FIR Implementation I:

```
./tFIR_implementationI -NB=100 -M=2048

->2048-th order FIR-> M forced to: 2048, No. of loops:4, with No. of threads:512
FIR order M: 2048 with NT 512 threads
No. of Blocks:100, where K:1, perBlock inp:4096 perBlock out:2049
        xlen:206848 , y_len:204801

Input file: ../../../projects/tFIR_implementationI/data/fir2DP.dat
Y_len = N-M+1 !!!
SUPERGOLD, casting to FLOAT according to device !
Data files would be too long , skipping.
Test PASSED
Processing time: 22.712999 (ms) and float MULs: 4.194324e+08
 MegaFlopsy: 18466.625000
```

Notes: While the FIR implementation is running, it's output is pretty verbose. One of important information printed out is the result of comparison of CUDA computation and clean C-code computation. If CUDA computation enumerates correct results, the string "Test PASSED" is printed out. Other important information is: `NT` - the number of threads executed on GPU, `perBlock inp` - amount of input vector numbers stored in shared memory on GPU that leads to `x_len` - the length of enumerated input vector which corresponds to input parameter, `NB`, number of blocks. Number of blocks and filter order gives the output vector length - `y_len`. Extended implementation (number I), also shows the extended configuration, e.g.:

```
->2000-th order FIR-> M forced to:  2048, No.  of loops:4, with No.  of threads:512
```

This configuration summary says that desired filter order, M=2000, is rounded to some multiple of 512, i.e. to 2048, and that there is need to compute 4 loops with 512-thread block.

Other information printed out is the `Processing time`, then the number of enumerated multiplication operations, `MULs`, and thus enumerated computational performance - `MegaFlops`.

B.2 Running the CAF implementation

MEX-CUDA implementation is ran from Matlab environment by m-script `cafCompare.m`.
Command-line output of MEX-CUDA implementation of the CAF:

```
N =
      131072
ns =
   300
L =
     1
m =
   600
*** Matlab routine ***
Elapsed time is 3.064718 seconds.
```

```
*** CUDA routine ***
Mex-CUDA Cross Ambiguity Function.
MEX: inputs are double precision
MEX: tau:600
MEX: L:1
MEX: NS:300
MEX: host_mem_size=2812kB
CUDA: N=131072 m=600 L=1 ns=300 NT=512 NB=256
device_mem_size=600MB for output
Elapsed time is 0.376213 seconds.
*** Error ***
ans =
     4.487918340601027e-04     7.843635074777922e-08
pisvejcConst =
   -65
cmin =
    -1.807462827090507e+02
cmax =
    -1.471502924317620e+02
```

CUDA run-time only implementation of the CAF is ran as any other CUDA SDK project, by it's binary `tCAF`.

Command-line output of CUDA run-time only CAF implementation:

```
./tCAF

CAF Exec: N:131072, tau 0->599, with batch size: 600 batches 1-times looped.
Kernel Exec: NT:512, NB:256, shmem_len:1536 of Complex; NL:1
Input file: ../../../projects/tCAF/cafdata/s1cplx.dat
Input file: ../../../projects/tCAF/cafdata/s2cplx.dat
Host code of Conjugate in 0.138000 (ms) and 1.310720e+05 ops => 949.797058 MFlops.
Check sizeof cufftComplex:8B vs. Complex:8B
stsizedev:629145600B , fftdatasize:629145600B
--->>>
Overall Processing time: 843.828003 (ms) and Ops: 1.415578e+09
 MegaFlopsy: 1677.566406
```

Notes: While the CAF implementation is running, it's output is pretty verbose. One of important information printed out is the comparison of Matlab routine reference results and CUDA routine results. Value of relative error enumerated is quoted by `*** Error ***` message followed by two numbers - maximum and minimum of relative error. Other important information printed out is the `Elapsed time`. Configuration of the CAF enumeration is also printed out for both routines. Following consideration parameters are printed out: the length of input vectors, `N`, and thus FFT-size, `m` - the number of enumerated delays, τ, `NS` - number of output samples used, `L` - number of main loops to be enumerated in case where the amount of GPU memory is less than necessary amount for all data. GPU execution configuration - threads and blocks, is also printed out.

C FIR filter on common CPU

Simple FIR implementation on PC for comparison with CUDA implementation was developed in ANSI C and compiled with gcc (GNU C compiler v4.1) with optimizations for the CPU type and math enabled e.g.: `gcc -march=nocona -O3 -ffast-math -funroll-loops ....` Numeric precision was set to single-precision (float type).

FIR enumeration loop stands like following piece of C-code for input vector x, output vector y, filter coefficients h with filter order M.

```
float sum, *x, *y, *h;

for (n=0; n<N; n++) {
  sum = 0;
  for ( m=0; m<M; m++) {
    sum += h[m] * x[M-m-1+n];
  }
  y[n] = sum;
}
```

Achieved performance of this FIR implementation at PC with 3GHz CPU was approximately 3GFlops with input sequence length 1M samples and FIR order M=33.

D CD-ROM content

Software package consists of four folders, each projects are:
tCAF - CAF implementation as CUDA runtime only.
tCAF-mex - CAF implementation as CUDA interfaced to Matlab.
tFIR_implementationI - FIR implementation no. I.
tFIR_implementationII - FIR implementation no. II.

```
CD-ROM:

tCAF
tCAF-mex
tFIR_implementationI
tFIR_implementationII

./tCAF:
cafdata
cucomplex.cu
cutesto.cu
Makefile
tCAF.cu
tCAF_kernel.cu

./tCAF/cafdata:
s1cplx.dat
s2cplx.dat

./tCAF-mex
cafCompare.m
caf_cuda.cu
caf_cuda.mexa64
cafdata.mat
caf_kernel.cu
caf_runtime.cu
cutil.h
cutil_inline.h
Makefile
nvmex
nvopts.sh

./tFIR_implementationI:
cutesto.cu
data
Makefile
tFIR.cu
tFIR_gold.cpp
tFIR_kernel.cu

./tFIR_implementationI/data:
fir2DP.dat
```

```
./tFIR_implementationII:
cutesto.cu
data
Makefile
tFIR.cu
tFIR_gold.cpp
tFIR_kernel.cu

./tFIR_implementationII/data:
fir2DP.dat
```

E Acknowledgement

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement No. 100029 and from the Czech Ministry of Education, Youth and Sport under grant agreement No. 7H09005.

References

[1] CUDA Compiler Driver NVCC. `http://www.nvidia.com/object/cuda_develop.html`.

[2] CUDA Programming Guide v2.2. `http://www.nvidia.com/object/cuda_develop.html`.

[3] CUDA Reference Manual. `http://www.nvidia.com/object/cuda_develop.html`.

[4] CUDA Zone. `http://www.nvidia.com/object/cuda_home.html`.

[5] Heřmánek, Antonín ; Kuneš, Michal ; Kvasnička, M. Comuputation of Long Time Cross Ambiguity function using reconfigurable HW. *Proceedings of the 6th IEEE International Symposium on Signal Processing and Information Technology. Vancouver.*, (ISBN 0-7803-9754-1):1–5., 2006.

[6] Kvasnička, M. ; Heřmánek, Antonín ; Kuneš, Michal. Implementace akcelerátoru pro výpočet pro výpočet věrohodnostní funkce [program]. Praha : ÚTIA AV ČR. CD ROM, 10,4 MB., 2007.

[7] Say Hello To DirectX 10, Or 128 ALUs In Action: NVIDIA GeForce 8800 GTX (G80). `http://www.digit-life.com/articles2/video/g80-part1.html`.

[8] Wikipedia. `http://en.wikipedia.org/wiki/CUDA`.