Accelerating batched 1D-FFT with a CUDA-capable computer

Calling CUDA library functions from a Java environment

R. de Beer and D. van Ormondt
Dept. of Applied Physics, Univ. of Technology Delft, NL
E-mail: r.debeer@tudelft.nl

F. Di Cesare and D. Graveron-Demilly
CREATIS-LRMN, Univ Lyon 1, CNRS UMR 5220
INSERM U630, INSA Lyon, FR
E-mail: Danielle.Graveron@univ-lyon1.fr

D.A. Karras
Chalkis Inst. of Technology, Dept. Automation,
Hellas, GR
E-mail: dakarras@ieee.org

Z. Starcuk
Dept. of Magnetic Resonance and Bioinformatics,
Inst. of Scientific Instruments of the ASCR, Brno, CR
E-mail: starcuk@ISIBrno.Cz

2010-05-16 12:19

***Index Terms*—Batched 1D-FFT, CUDA-enabled GPU, CUFFT library, Java bindings, home-assembled PC, *j*MRUI software package, exhaustive search in MRS**

I. Introduction

This work concerns the application of CUDA-based software (Compute Unified Device Architecture), developed by NVIDIA for programmable Graphics Processing units (GPUs) [1]. CUDA code is written in 'C for CUDA', indicating the standard C programming language with NVIDIA extensions. The advantage of using CUDA is that one can accelerate numerical computations, traditionally handled by Central Processing Units (CPUs), by CUDA-enabled GPU devices, particularly if the numerical problems at hand are suited for parallel computing.

A. Our goal

Our goal was to find out, whether batched (multiple) one-dimensional Fast Fourier Transformation (1D-FFT), often encountered in various fields of signal processing, can be speeded up significantly by exploiting the parellel-processing power of a low-cost, standard, CUDA-enabled graphics card in a home-assembled PC.

Batched 1D-FFT is of particular interest to us for the following reasons:

- It is applied extensively in the Java Magnetic Resonance User Interface (*j*MRUI) software package of our 'FAST' European Union project[2].
- In our recent work on handling unknown Magnetic Resonance Spectroscopy (MRS) lineshapes, batched 1D-FFT plays an essential role when applying exhaustive search in a semi-parametric, two-criterion, NLLS fit of the MRS parameters [3].

Since the GUI of our *j*MRUI software package has been written in Java, we want to call relevant CUDA library fuctions from a Java environment. To that end the Java bindings for CUDA approach of jcuda.org [4], based on the Java Native Interface (JNI), was employed.

Ultimately, we want to embed the Java-bindings based calls to CUDA into a plug-in for the *j*MRUI platform [5].

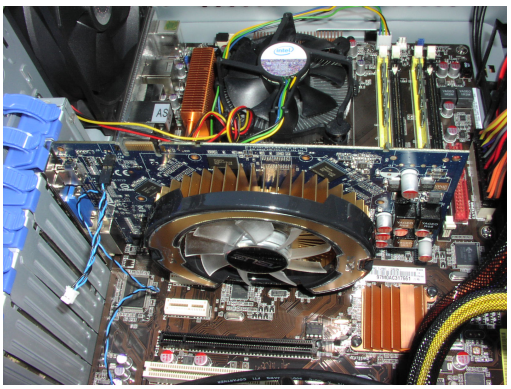

Figure 1. NVIDIA *GeForce 9600 GT* -based graphics card (CUDA-enabled) in a home-assembled desktop PC.

II. Methods

The methods applied can be devided into hardware- and software-based. The hardware concerns a home-made assemblage of a low-cost CUDA-capable desktop PC. It's essential parts are mentioned in II-A.The software part, described in II-B, concerns the installation of the CUDA software [1] and, in turn, the installation of the Java bindings for CUDA software package [4].

A. Hardware

In order to get an impression of the cost and time involved in building a CUDA-capable computer, we have

assembled a low-cost desktop PC. The parts, chosen, reflect the PC state-of-the-art of about $1\frac{1}{2}$ years ago. We like to mention the following essential parts:

- An ASUS *P5Q PRO Turbo* motherboard with Intel *Core 2 Duo E8400* CPU and 4 GB RAM.
- An ASUS *EN9600GT/DI/512MD3* graphics card with NVIDIA *GeForce 9600 GT* chipset (CUDA-enabled GPU) and 512 MB DDR3 memory (see Figure 1).

B. Software

1) Operating system: The home-assembled desktop PC is equipped with the Ubuntu 9.10 (32-bit version) Linux operating system. It was installed from a running live CD, which gave the opportunity of first checking the hardware. The basic installation took about one hour (applying a software update by the Update Manager included).

2) Installing a certified Linux NVIDIA driver: It is important for the performance of a CUDA-capable Linux system that one has installed a *recent* version of the Linux NVIDIA driver. We have chosen to work with the latest certified NVIDIA driver, which at the time of this study was version 190.42.

3) Installing the CUDA software: The CUDA Development Tools for Linux 32-bit operating systems, that we have used, had release number 2.3. They include two parts, called the CUDA Toolkit and the CUDA SDK respectively. The Toolkit contains the tools to build and compile CUDA applications and the SDK contains code samples. Their installer files (shell scrips) can be downloaded from the CUDA website [1].

The CUDA Development Tools should be installed by running their installer shell scrips. Before being able to compile and execute CUDA applications, one should take care of the required inclusions for the environment variables PATH and LD_LIBRARY_PATH.

4) Installing the Java bindings for CUDA software: In order to provide access to the CUDA software from a Java environment, we have downloaded and installed the Java bindings for CUDA software package (JCuda). Its download archive file contains all JAR files and library SOs required for 32-bit Linux [4].

An important aspect of installing the JCuda software is to provide the correct Java CLASSPATH to the JCuda JAR files. We found it the easiest approach of using the `-classpath` option, when calling the Java programming language compiler (`javac`) and Java application launcher (`java`).

III. Results

A. Assembling the CUDA-capable desktop PC

The cost of the CUDA-capable desktop PC (the monitor excluded) amounted to about €600. Its total (summed up) assemblage time was of the order of a few days.

B. Running the CUDA SDK test programs

In order to verify the CUDA installation, one should run the CUDA SDK test programs `deviceQuery` and `bandwidthTest`.

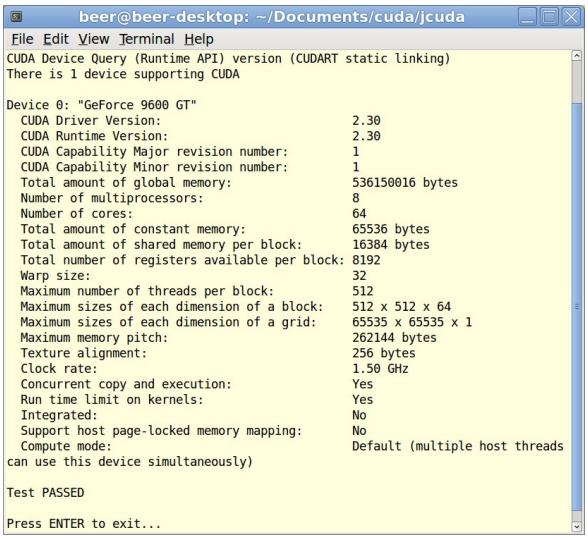

Figure 2. Valid result for the CUDA `deviceQuery` test program.

The result for the `deviceQuery` test on our system is shown in Figure 2. When comparing with the result in the CUDA Getting Started manual [6] it can be seen that it is a valid result for the NVIDIA *GeForce 9600 GT* device. The output also shows the fewer potentials of *GeForce 9600 GT* with respect to more recent NVIDIA GeForce devices.

C. Performing CUDA-based batched 1D-FFT within a Java environment

In order to test the computation time of CUDA-based batched (multiple) 1D-FFT, when called from a Java program, we took as a starting point the *JCufftSample.java* code from the jcuda.org website [4]. In that sample Java code a CUFFT-library [7] -based complex-to-complex forward 1D-FFT is executed and compared with the result obtained from the Java `JTransforms` [8] approach.

Since in the original *JCufftSample.java* program the `batch` input parameter for the CUFFT 1D-FFT initialization (function `cufftPlan1d`) is set at 1, we could not test the opportunity of performing batched 1D-FFT in a *parallel* fashion [7]. We therefore have extended the code, particularly by introducing a series (batch) of 1D input signals for the 1D-FFT (for `JTransforms` as well as for CUFFT) and by introducing timing code.

The essential (generalized) CUDA part of the extended source code can be seen in Figure 3. Note that we have included timing for `cufftPlan1d` as well as for the actual 1D-FFT execution (function `cufftExecC2C`).

```
.............................
int batch = 1024;
int size = 1024;
long timing1 = 0;
long timing2 = 0;
float data[][];
data = new float[batch][size*2];
float tmp[];
tmp = new float[batch*size*2];
.............................
..for looping over JTransforms...
.............................
for (int i=0; i<batch; i++)
{
  for (int j=0; j<size*2; j++)
  {
    tmp[i*size*2+j] = data[i][j];
  }
}
cufftHandle plan = new cufftHandle();
long time1 = System.nanoTime();
JCufft.cufftPlan1d(plan, size,
       cufftType.CUFFT_C2C, batch);
long time2 = System.nanoTime();
JCufft.cufftExecC2C(plan, tmp, tmp,
       JCufft.CUFFT_FORWARD);
long time3 = System.nanoTime();
timing1 = time2 - time1;
timing2 = time3 - time2;
for (int i=0; i<batch; i++)
{
  for (int j=0; j<size*2; j++)
  {
    data[i][j] = tmp[i*size*2+j];
  }
}
JCufft.cufftDestroy(plan);
.............................
```

Figure 3. Snippet of extended code (generalized, essential CUDA part) of the *JCufftSample.java* sample program [4]. The grey colorboxes denote the actual JCuda calls to the CUFFT library. Note the `batch` parameter.

Table 1: Elapsed time (in msec) for batched `JTransforms` (not shown in the code of Figure 3) and batched `JCufft.cufftExecC2C` as a function of `batch` and `size` (both parameters were equalized). Also the `JCufft` 1D-FFT initialization time (`JCufft.cufftPlan1d`) is given. Finally, `factor` denotes the ratio of the batched `JTransforms` and batched `JCufft.cufftExecC2C` times.

	256	512	1024	2048
JTransforms	38	199	351	385
JCufft.cufftExecC2C	8	9	16	36
JCufft.cufftPlan1d	65	64	64	64
factor	5	22	22	11

In Table 1 the benchmark results for batched 1D-FFT via `JTransforms` and via `JCufft.cufftExecC2C` as a function of the input parameters `batch` and `size` are given. In these tests both parameters were set equal to each other. Note, that although the timing function `System.nanotime()` in Figure 3 suggests a very high time resolution, the accuracy of the resulting timing values appeared to be much less.

D. Performing batched 1D-FFT with 'C for CUDA' code

In order to get an impression of the overhead, caused by calling CUDA functions from a Java environment (as was done for getting the results described in III-C), we repeated the CUDA part of the benchmark with code written directly in 'C for CUDA'.

```
.............................
int batch = 1024;
int size = 1024;
struct timeval tv_start, tv_stop;
cufftComplex *data_h, *data_d;
size_t memsize = batch*size*sizeof
       (cufftComplex);
cudaMallocHost((void**)&data_h, memsize);
cudaMalloc((void**)&data_d, memsize);
.....insert data into data_h.....
cufftHandle plan;
cudaMemcpy(data_d, data_h, memsize,
       cudaMemcpyHostToDevice);
cufftPlan1d(&plan, size, CUFFT_C2C, batch);
gettimeofday(&tv_start, NULL);
cufftExecC2C(plan, data_d, data_d,
       CUFFT_FORWARD);
gettimeofday(&tv_stop, NULL);
double timing_tv = (double)
     (tv_stop.tv_usec - tv_start.tv_usec)/1000.0;
cudaMemcpy(data_h, data_d, memsize,
       cudaMemcpyDeviceToHost);
.....extract data from data_h....
cufftDestroy(plan);
cudaFreeHost(data_h);
cudaFree(data_d);
.............................
```

Figure 4. Snippet of 'C for CUDA' code (generalized) for batched 1D-FFT. The grey colorboxes denote the actual CUDA calls. For the sake of simplicity only one timing is shown. Note again the `batch` parameter.

In Figure 4 the (generalized) code for the 'C for CUDA'-based benchmark is presented. Compared to Figure 3 there are more lines of code since memory allocation/freeing (on host and CUDA device) as well as data transfer between host and device memory now explicitly must be stated.

Table 2: Elapsed time (in msec) for batched `cufftExecC2C` as a function of `batch` and `size` (again both parameters were equalized). Note the lack of the `JCufft` naming part, when compared to Table 1, indicating that now 'C for CUDA' code was used for the benchmark. Again also the 1D-FFT initialization time (`cufftPlan1d`) is given. Finally, the data-transfer times to and from the CUDA device (`cudaMemcpy`) are given.

	256	512	1024	2084
cufftExecC2C	0.08	0.08	0.13	0.15
cufftPlan1d	0.04	0.04	0.06	0.06
cudaMemcpyHostToDevice	0.1	0.4	1.7	6.2
cudaMemcpyDeviceToHost	0.2	0.7	3.2	14.0

Table 3: Data-transfer bandwidth (in GB/sec) for `cudaMemcpy` as a function of `batch` and `size` (both parameters were equalized).

	256	512	1024	2084
cudaMemcpyHostToDevice	4.2	4.6	4.7	5.0
cudaMemcpyDeviceToHost	2.4	2.8	2.5	2.2

In Table 3 the data-transfer bandwidth (the rate) for `cudaMemcpy` (to and from the CUDA device) is presented, as calculated from

$$\text{bandwidth} = \frac{1000 \times \text{memsize}}{\text{time} \times 1024^3}, \qquad (1)$$

where bandwidth is in GB/sec, memsize is the number of bytes of the batched, complex-valued, float, data array and time is the elapsed time (timing) in msec. The bandwidth quantity often is an important factor for measuring *performance* in the field of GPU computing [9].

E. User-guided exhaustive search in MRS parameter space

When performing *in vivo* quantitation of metabolites in MRS, one ususally applies some form of nonlinear least-squares (NLLS) fitting of a physical model function to the MRS data. In order to handle unknown *lineshapes*, i.e. unknown *decays* in the measurement (time) domain, we recently [3] have introduced a second NLLS criterion, based on the general prior knowledge that the width of the lineshape is limited. In the results, presented in this subsection, we have replaced this second NLLS optimization step by an exhaustive search of the relevant MRS parameters.

The method has as a starting point [3], that under certain conditions an *in vivo* MRS signal $s(t)$ can be modelled by

$$s(t) = \text{decay}(t)\hat{s}_{\text{nodecay}}(t) + \text{noise}(t), \qquad (2)$$

where $\text{decay}(t)$ is the (usually unknown) decay function and $\hat{s}_{\text{nodecay}}(t)$ the *a priori* known non-decaying version of the model function, obtained by summing over the contributions from the individual metabolites. In this context the $\hat{}$ on $\hat{s}_{\text{nodecay}}(t)$ indicates a known mathematical function with unknown values of the parameters.

In the code listed below (see Figure 5) we have illustrated the example case of having three MRS parameters (amplitudes) a_1, a_2 and a_3, representing the concentrations of three metabolites. Since in the exhaustive search only the ratio of the amplitudes can be determined, the third parameter a_3 was kept fixed at 1.0 and the values of the others were varied. The starting ratio was assumed to be found by a separate round of the first NLLS criterium.

After applying a simple estimator for $\mathsf{decay}(t)$ [3] (for all values of a_1 and a_2 in the exhaustive-search grid), an essential next step in the method is to perform a batched 1D-FFT of all $n_1^{max} \times n_2^{max}$ decay functions. Finally, the (a_1,a_2)-combination is to be determined at which the quantity $|\text{Re:FFT}[\mathsf{decay}(t)]|$ summed over the region $|\nu| > \nu_{threshold}$ is minimal.

```
.............................
int n1max = 90;
int n2max = 90;
int n3max = 1;

int batch = n1max*n2max*n3max;
int ndp = 1024;

int n1mul = n2max*n3max*ndp;
int n2mul = n3max*ndp;
int n3mul = ndp;

float summin = 1000000.0;

float a1start = 0.25;
float a2start = 0.50;
float a3start = 1.00;

float a1step = 0.0001;
float a2step = 0.0001;
float a3step = 0.0;

float sum;
float *a1, *a2, *a3, *modlshape;

cufftComplex decay, *data_h;

......other initializations......
.........and assignments.........

for (n1=0; n1<n1max; n1++)
{
  a1[n1] = a1start - (float)
      0.5*n1max*a1step + (float) n1*a1step;
  for (n2=0; n2<n2max; n2++)
  {
    a2[n2] = a2start - (float)
        0.5*n2max*a2step + (float) n2*a2step;
    for (n3=0; n3<n3max; n3++)
    {
      a3[n3] = a3start - (float)
          0.5*n3max*a3step + (float) n3*a3step;
      for (n=0; n<ndp; n++)
      {

        ...estimate decay from MRS signal...

        index = n1*n1mul + n2*n2mul + n3*n3mul
          + n;
        data_h[index].x = decay.x;
        data_h[index].y = decay.y;
      }
    }
  }
}

....batched 1D-FFT via cufft.....

for (n1=0; n1<n1max; n1++)
  for (n2=0; n2<n2max; n2++)
    for (n3=0; n3<n3max; n3++)
```

```
{
  sum = 0.0;
  for (n=0; n<ndp; n++)
  {
    index = n1*n1mul + n2*n2mul + n3*n3mul
    + n;
    modlshape[n] =
        sqrtf(data_h[index].x
        *data_h[index].x);

    if ((n > (int)(0.2*(float)ndp)) &
        (n < (int)(0.8*(float) ndp)))
    {
      sum = sum + modlshape[n];
    }
  }

  if (sum < summin)
  {
    summin = sum;
    n1tune = n1;
    n2tune = n2;
    n3tune = n3;
    a1tune = a1[n1tune];
    a2tune = a2[n2tune];
    a3tune = a3[n3tune];
  }
}
.................................
```

Figure 5. Snippet of 'C for CUDA' code (generalized) for user-guided exhaustive search in MRS parameter space. The grey colorboxes denote CUFFT-based code (see Figure 4). For the sake of simplicity timings are not shown now.

Table 4: Elapsed time (in msec) and data-transfer bandwidth (in GB/sec) for various parts of the exhaustive-search 'C for CUDA' code in Figure 5 (`batch` = 8100 and `ndp` = 1024).

	time	bandwidth
`insert data into data_h`	417	
`cudaMemcpyHostToDevice`	12	5.0
`cufftPlan1d`	0.06	
`cufftExecC2C`	0.18	
`cudaMemcpyDeviceToHost`	18	3.4
`extract data from data_h`	249	

In Table 4 the benchmark results for various parts of the exhaustive-search 'C for CUDA' code in Figure 5 are shown. In this sample case we have worked with the simulated MRS signal (noiseless version) used in our lineshape study [3].

Figure 6 gives an impression of the minimizing of |Re:FFT[decay(t)]| (summed over $|\nu| > \nu_{threshold}$), when varying (a_1,a_2) (the third amplitude a_3 was kept fixed at 1.0). We have found an amplitude ratio of 0.2498 : 0.4994 : 1.0, which agrees well with the true ratio of 0.25 : 0.5 : 1.0.

Finally we like to note that we have called this subsection '*User-guided* exhaustive search' because, firstly, we have assumed to have a reasonably good starting ratio for the amplitudes (obtained with a separate round of the first NLLS criterium) and, secondly, we have established in a user-interactive way a suited grid for (a_1,a_2).

IV. Discussion

A. Concerning the Java-based benchmark

The numbers in Table 1 show that the `JCufft` initialization time is larger than the batched `JCufft` execution time itself. Moreover, the initialization time is independent of the `batch` and `size` parameters (for the given range). This suggests that acceleration of batched `JCufft` with respect to batched `JTransforms`, as indicated by `factor` in Table 1, is realized only if batched `JCufft` is performed several times with using the same `JCufft` initialization.

Concerning the acceleration factor of 22 in Table 1 (for `batch` = 512, 1024) we like to mention that this result more or less agrees with results that can be deduced from tables in [8] (for the case of single 1D-FFT). Moreover, in the latter benchmark it also is shown that `JTranforms` has comparable execution times with respect to FFTW [10]. This is of interest to us since the FFTW method is applied in our *j*MRUI software package.

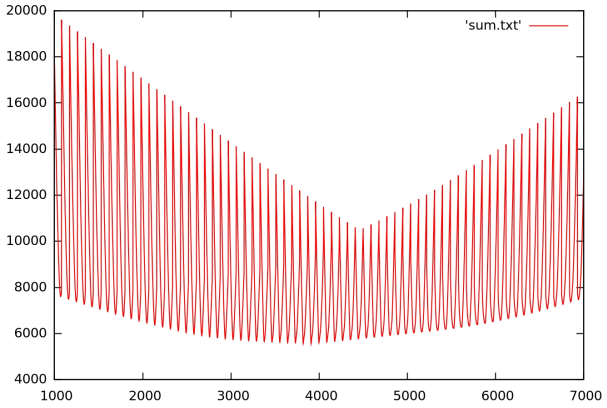

Figure 6. The quantity |Re:FFT[decay(t)]| as a function of `index` (see text and Figure 5).

B. Concerning the 'C for CUDA'-based benchmark

When comparing Table 1 with 2, a striking difference in 1D-FFT initialization time (see timings of the `cufftPlan1d`'s) seems to be indicated. However, from inspection of the `JCufft` Java class (see source code [4]) it can be learned that its `cufftPlan1D` method performs dynamic library loading (library initialization), if it is the *first* time a `JCufft` method is being called. We have investigated this aspect any further in a separate code, in which the `JCufft.cufftPlan1D` method was called twice. The timing found for the second call yielded a negligible small value, in agreement with the 'C for CUDA'-results for `cufftPlan1D` listed in Table 2.

Another point of interest in the Table 1 - 2 comparison is the difference in `cufftExecC2C` timings. Again here the explanation can be found in

the source code of the `JCufft` Java class. It appears that memory allocation/freeing and data transfer (between host and CUDA device), as mentioned in III-D, are included in the `JCufft.cufftExecC2C` method. This is clearly demonstrated by the fact that the summed timings for `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` (in Table 2) are of the same order of magnitude as the timings for `JCufft.cufftExecC2C` (in Table 1).

Finally we like to note that the data-transfer bandwidths, presented in Table 3, are of the same order of magnitude as the 8 GB/sec bandwidth reported for the PCI Express 2.0×16 bus [9] (used in our desktop PC).

C. Concerning the exhaustive-search benchmark

From inspection of Table 4 it can be seen that inserting data into or extracting from `data_h` (the array involved in the batched 1D-FFT) by far is the dominant timing factor. This clearly is caused by the two fourfold for-statement loopings involved (see Figure 5). This suggests that these code parts also must be carried out in the CUDA device.

V. Concluding remarks

Summarizing we like to make the following concluding remarks:

- We have assembled a low-cost CUDA-capable desktop PC, reflecting the PC state-of-the-art of about $1\frac{1}{2}$ years ago.
- Via the Ubuntu 9.10 Linux operating system we could enable CUDA by installing a recent Linux NVIDIA driver and the CUDA software (version 2.3).
- By applying the Java-bindings based JCuda software package we could call CUFFT library functions from a Java environment.
- We could easily perform batched (multiple) 1D-FFT in a parallel fashion by exploiting the batch facility of CUFFT 1D-FFT for a CUDA-enabled GPU device. In this way we could avoid for statement looping, needed for the (CPU-based) reference method.
- We could speed up the batched 1D-FFT execution time by about a factor of 20 by applying the GPU-based rather than the CPU-based approach.
- Easy comparison of Java-based and 'C for CUDA'-based benchmarking appeared to be hindered by the choices made for the JCuda implementation.
- The CUDA-based benchmark results, reported in this work, seemed to be limited by the data-transfer bandwidth of the computer PCI Express 2.0×16 bus.
- If data-transfer speed indeed is the limiting factor, significant computational accelerations can only be achieved if major parts of the numerical calculations can be carried out in the CUDA GPUs.
- In the context of the latter, enhanced double-precision and amount of local memory of recent/future CUDA devices will become important.
- Using CUDA-based batched 1D-FFT, we could carry out a sample user-guided exhaustive-search in MRS parameter space.

Acknowledgment

This work is supported by Marie-Curie Research Training Network 'FAST' (MRTNCT-2006-035801, 2006-2009).

References

[1] NVIDIA, "CUDA 2.3 downloads," http://developer.nvidia.com/object/cuda_2_3_downloads.html, 2009.

[2] D. Stefan, F. D. Cesare, A. Andrasescu, E. Popa, A. Lazariev, E. Vescovo, O. Strbak, S. Williams, Z. Starcuk, M. Cabanas, D. van Ormondt, and D. Graveron-Demilly, "Quantitation of magnetic resonance spectroscopy signals: the jMRUI software package," *Meas. Sci. Technol.*, vol. 20, p. 104035 (9pp), 2009.

[3] E. Popa, E. Capobianco, R. de Beer, D. van Ormondt, and D. Graveron-Demilly, "In vivo quantitation of metabolites with an incomplete model function," *Meas. Sci. Technol.*, vol. 20, p. 104032 (9pp), 2009.

[4] Jcuda.org, "Java bindings for CUDA," http://www.jcuda.org/, 2010.

[5] D. Stefan, A. Andrasecu, E. Popa, H. Rabeson, O. Strbak, Z. Starcuk, M. Cabanas, D. van Ormondt, and D. Graveron-Demilly, "jMRUI Version 4 : A Plug-in Platform," in *IEEE International Workshop on Imaging Systems and Techniques, IST 2008*, Chania, Greece, 10-12 September 2008, pp. 346–348.

[6] NVIDIA, "CUDA Getting Started, version 2.3," http://developer.nvidia.com/object/cuda_2_3_downloads.html, 2009, manual.

[7] ——, "CUDA CUFFT Library, version 2.3," http://developer.nvidia.com/object/cuda_2_3_downloads.html, 2009, manual.

[8] P. Wendykier, "JTransforms," http://sites.google.com/site/piotrwendykier/software/jtransforms, 2009, an open source multithreaded FFT library written in pure Java.

[9] NVIDIA, "CUDA Best Practices Guide, version 2.3," http://developer.nvidia.com/object/cuda_2_3_downloads.html, 2009, manual.

[10] M. Frigo and S.G. Johnson, "FFTW," http://www.fftw.org/, 2010, a C subroutine library for computing the discrete Fourier transform.