Numerical computation of Coulomb potential

Konrad Kobuszewski

April 30, 2017

1 Naive method. Accurancy problems.

We consider continuous distribution of charge $\rho(\boldsymbol{r}) = qn(\boldsymbol{r})$, where $n(\boldsymbol{r}) = \sum_{k=1}^{N} |\psi_k|^2(\boldsymbol{r})$ ($\psi_k \in \mathbb{C}$ for $N = 1$ is assumed to be the input of the program). Coulomb potential of such a distribution is given by (we used $\varepsilon_0 = 1$):

$$V_{coulomb}(\boldsymbol{r}) = \frac{1}{4\pi}\int_{\mathbb{R}^3} \frac{q^2}{|\boldsymbol{r}-\boldsymbol{r}'|} n(\boldsymbol{r}')\, d^3r' = \mathcal{F}^{-1}\left[\frac{q^2}{k^2}\mathcal{F}[n(\boldsymbol{r})](\boldsymbol{k})\right](\boldsymbol{r})$$

This is general solution of Poisson equation in an integral form.
We used Borel's convolution theorem for conversion in equation above.
In case of testing accurancy of algorithm $n(\boldsymbol{r}) = \frac{1}{(2\pi\sigma^2)^{3/2}} e^{-\frac{r^2}{2\sigma^2}} = \frac{1}{\left(\pi a_{ho}^2\right)^{3/2}} e^{-\frac{r^2}{a_{ho}^2}}$ will be used, because analitic solution is known (wiki):

$$V_{coulomb}(\boldsymbol{r}) = \frac{1}{4\pi}\frac{q^2}{r}\operatorname{erf}\left(\frac{r}{\sqrt{2}\sigma}\right) = \frac{1}{4\pi}\frac{q^2}{r}\operatorname{erf}\left(\frac{r}{a_{ho}}\right)$$

$a_{ho} = \sqrt{\frac{\hbar}{m\omega}}$ is characteristic length of harmonic oscilator (groundstate wavefunction is a kind of gaussian).

Advantages:

- Straight forward implementation
- Analytical for periodic case

Disadvantages:

- Due to long range behaviour of Coulomb potential poor accurancy in nonperiodic case is expected.

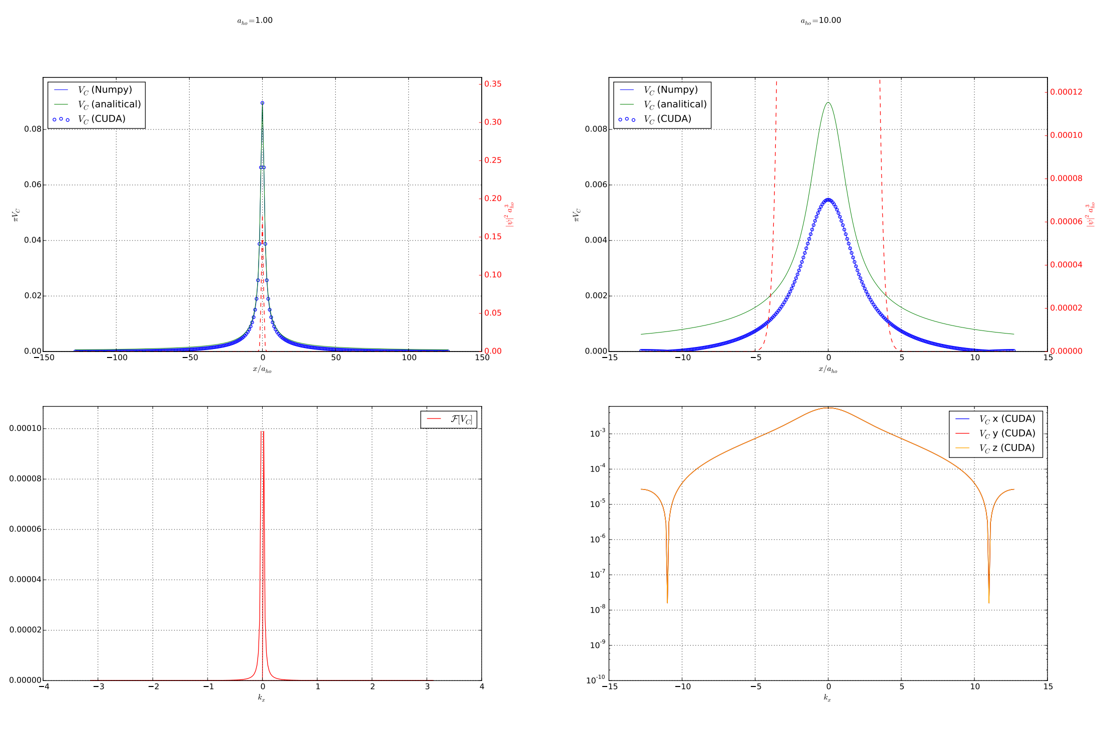

Figure 1: Ilustration of misaccurancy of computing Coulomb potential with naive method.

2 Methods of increasing accuracy of Coulomb potential computation

2.1 Performing computation on larger lattice

First conclusion from analysis of fig. 1 suggests just to perform computation on bigger grid.
We need to get values from smaller lattice and copy to „centre” of bigger array and also fill other entries of bigger array with zeros.

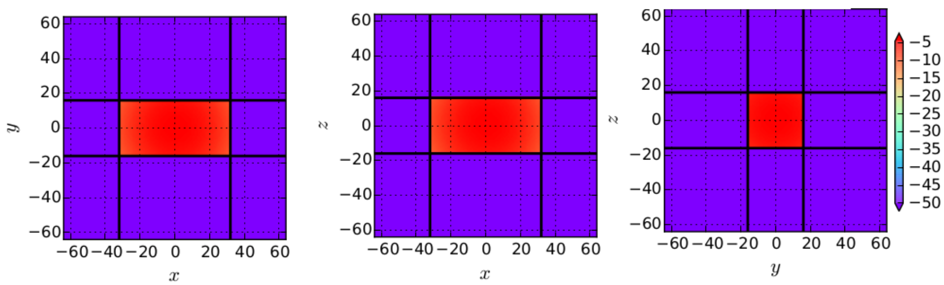

Figure 2: Example of resizing array in 3D. Colors correspond to density in logscale (orginal array was filled with constant).

Increase of lattice size in fact gives no more information about charge distribution, so there could exists better way for increasing accuracy without unnecessary increase of lattice and computation time, nevermore little more math have to be used. Two different cutoff methods dealing with this issue will be presented in next sections.

2.2 Spherical cutoff method

Given a cubic lattice of size $\left(1+\sqrt{3}\right)L_c$

$$\mathcal{F}^{-1}\left[\frac{q^2}{k^2}\left(1-\cos\left(\sqrt{3}L_c k\right)\right)\mathcal{F}\left[n\left(\boldsymbol{r}\right)\right]\left(\boldsymbol{k}\right)\right]\left(\boldsymbol{r}\right)$$

To deal with division by 0 we can use$\lim_{k\to 0}\frac{q^2}{k^2}\left(1-\cos\left(\sqrt{3}L_c k\right)\right)=\frac{3}{2}q^2L_c$.

Advantages:

- Accurate.

Disadvantages:

- The lattice is much bigger, performing cuFFT and vector-vector multiplication is getting slower.

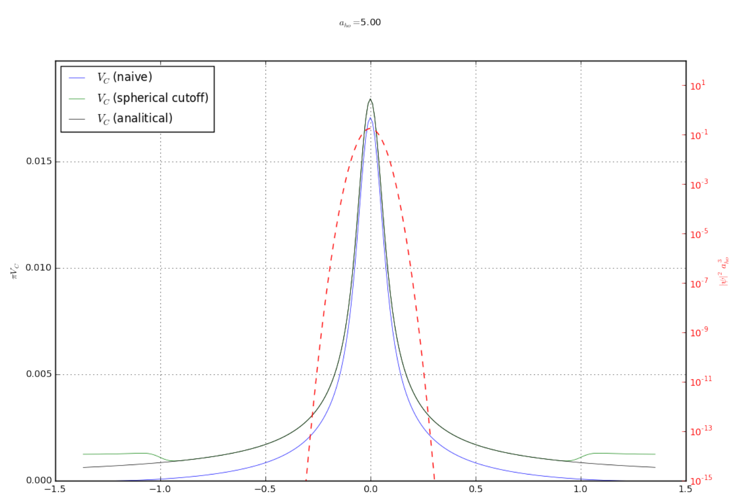

Figure 3: Spherical cutoff method is expected to give accurate results except edges of the lattice, so bigger box for computation and truncation of results is needed. (the red plot is density of charge in log-scale)

2.3 Cubic cutoff method

(not done yet...)

3 Performance tests

3.1 Comparison on device

We implemented and tested 3 types of functions counting number of different entries in two arrays (for comparision purpose). Each function returns number of entries with absolute difference *eps* between same entries in each array.
First solution was based on **thrust::inner_product** and two another use **__syncthreads_count** for blockwise comparison and custom reductions for suming results of each blocks. kernel_compare is kernel using every thread for comparision (like in simple reduction) and kernel_compare_multipass is based on thread fence reductions in Nvidia CUDA Samples. Both functions sum over blocks using custom reduction from Nvidia CUDA Samples (Harris example). The code is in files “compare.cuh”, “reductions.cuh” and “compare_thrust.cuh”.
Example of results given by nvprof (comparision of arrays of 307712 double elements, 100 calls for each function):

Time(%)	Time	Calls	Avg	Min	Max	Name
29.27%	5.8410ms	100	58.410us	56.958us	74.909us	void kernel_compare_multipass
29.22%	5.8318ms	100	58.317us	56.350us	71.325us	<thrust part 1>
26.72%	5.3325ms	100	53.324us	51.870us	66.237us	void kernel_compare<double>
4.42%	882.08us	100	8.8200us	8.4480us	10.367us	<thrust part 2>
3.72%	742.12us	303	2.4490us	2.2080us	7.3280us	[CUDA memcpy DtoH]
3.04%	607.33us	200	3.0360us	2.5280us	3.2960us	void __reduce_kernel__

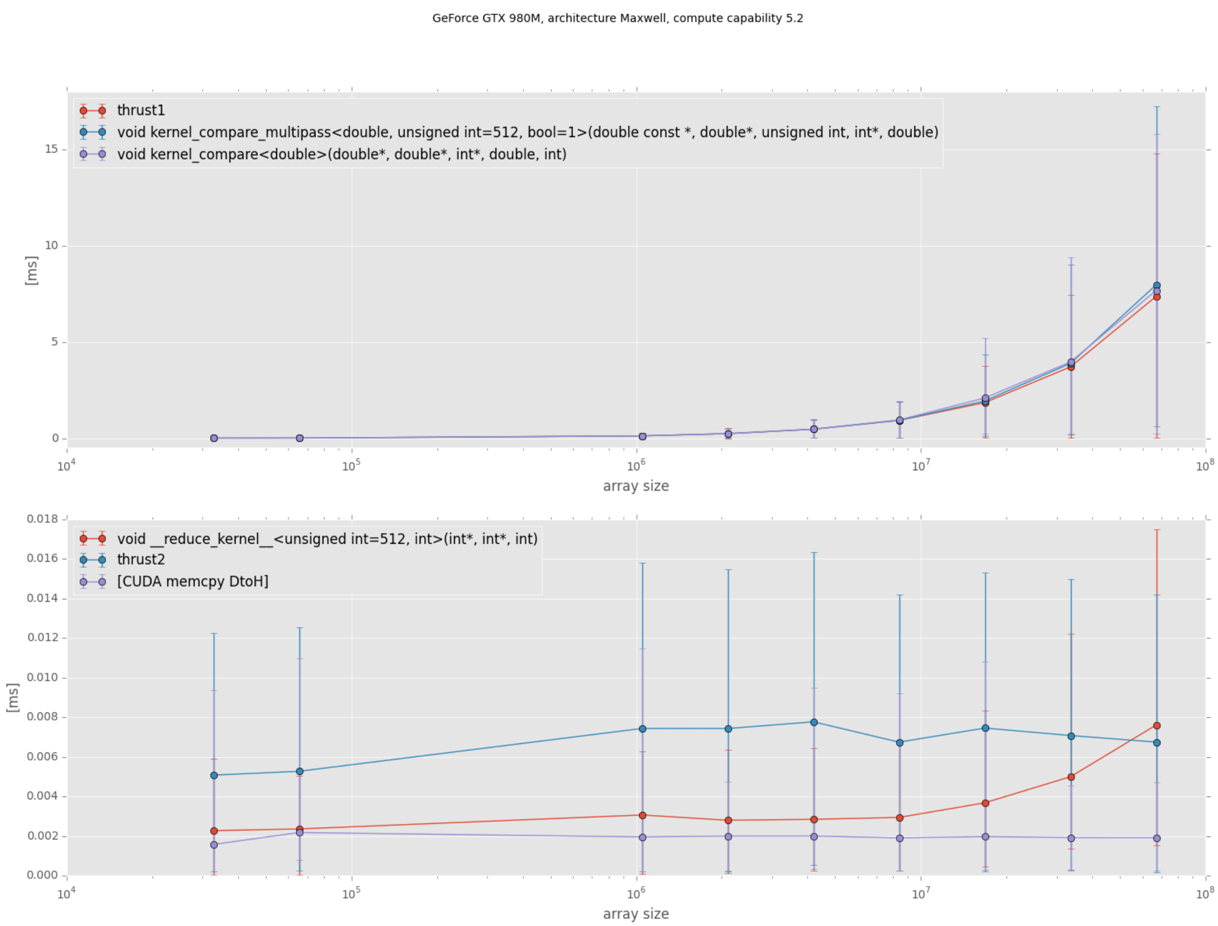

Figure 4: Results from nvprof for different sizes of input array. Results of custom reductions and memcpy should be added to custom comparisons and timings for thrust1 and thrust2 kernels should also be taken together. Solution based on thrust is comparable for samll arrays and noticeably faster for bigger arrays.

Comparison reveals that we could be able to improve the implementation based on **thrust::inner_product** by custom kernels, so this kernel will be used in further part of the project.

3.2 CUFFT complex-to-complex and real-to-complex

Output from nvprof for complex-to-complex transfroms (lattice 256x256x256, GeForce GTX 980M):

Time(%)	Time	Calls	Avg	Min	Max	Name
29.97%	42.821 ms	2	21.410ms	21.399ms	21.422ms	[CUDA memcpy DtoH]
18.52%	26.457 ms	2	13.229ms	13.220ms	13.237ms	void dpRadix0256C::kernel1Mem
18.51%	26.451 ms	2	13.225ms	13.212ms	13.238ms	void dpRadix0256C::kernel1Mem
15.23%	21.762 ms	5	4.3524ms	928ns	21.758ms	[CUDA memcpy HtoD]
6.78%	9.6858 ms	1	9.6858ms	9.6858ms	9.6858ms	void dpVector0256C::kernelMem
6.78%	9.6832 ms	1	9.6832ms	9.6832ms	9.6832ms	void dpVector0256C::kernelMem
4.21%	6.0161 ms	1	6.0161ms	6.0161ms	6.0161ms	kernel_vcoulomb

Output from nvprof for real-to-complex and complex-to-real transfroms (lattice 256x256x256, GeForce GTX 980M):

Time(%)	Time	Calls	Avg	Min	Max	Name
28.08%	22.014 ms	2	11.007ms	10.933ms	11.081ms	[CUDA memcpy DtoH]
16.74%	13.122 ms	2	6.5608ms	6.5487ms	6.5730ms	void dpRadix0256C::kernel1Tex
16.71%	13.100 ms	2	6.5499ms	6.5469ms	6.5529ms	void dpRadix0256C::kernel1Tex
13.82%	10.836 ms	5	2.1673ms	896ns	10.833ms	[CUDA memcpy HtoD]
5.40%	4.2312 ms	1	4.2312ms	4.2312ms	4.2312ms	<__nv_static_callback_t>
5.01%	3.9293 ms	1	3.9293ms	3.9293ms	3.9293ms	void dpVector0128C::kernelTex
5.01%	3.9267 ms	1	3.9267ms	3.9267ms	3.9267ms	void dpVector0128C::kernelTex
4.95%	3.8820 ms	1	3.8820ms	3.8820ms	3.8820ms	<__nv_static_callback_t>
4.29%	3.3631 ms	1	3.3631ms	3.3631ms	3.3631ms	kernel_coulomb_real

Using real-to-complex versions of transfroms gives about 40% speedup and requires nearly only half of the memory needed by complex-to-complex transform. According to this observations using real-to-complex versions is a necessity.
Coulmb kernel (naive, inaccurate implementation) takes only about 10% of total computation time, so its optimization will not give significant speed up for whole algorithm. Nevertheless real-to-complex transfrom rewrites array of [Nx,Ny,Nz] double entries to array [Nx,Ny,Nz/2+1] cuDoubleComplex entries, so memory alignment have to be considered more carefully.

3.3 Coulomb kernel

3.3.1 Preliminaries

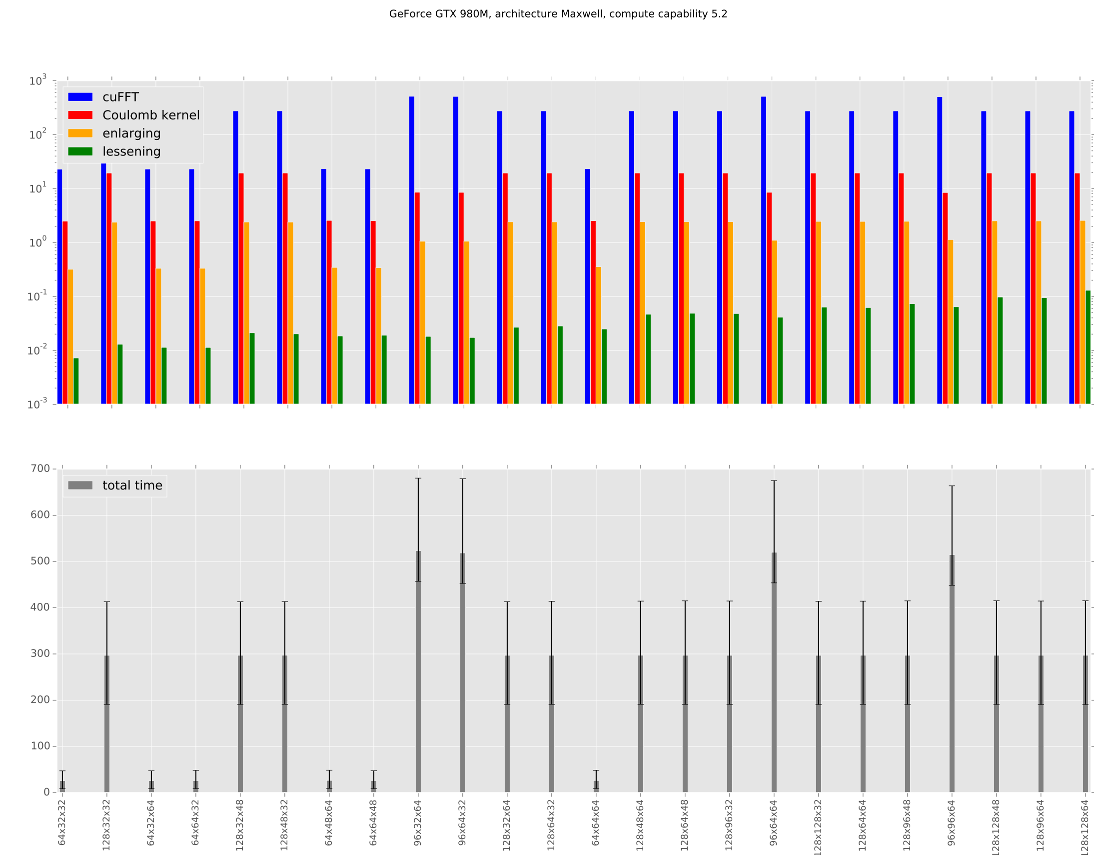

Figure 5: Performace tests for non-optimized implementation with power of 2 dimensions. Note that upper figure is in logscale and the resized array is cube proportional to the largest dimension, so sometimes smaller input array has worse performance.

- After brief perfomance analysis of parts of algorithm we found out that cuFFT and Coulomb convolution kernel take of an order of magnitude more time than resizing operations on arrays.
- Calls of cuFFT cannot be optimized further, so the only performance issue is usage of real-to-complex transforms. Investigation of cuFFT callback feature also doesn't seem to gain speed at all...
- Probably the only part of algorithm that can be impoved is Coulomb convolution kernel, so most of affort was spent on this optimization.

3.3.2 Memory alignment

We can use resizing of arrays to align memory properly and avoid using strided cuFFT, which is fairly complicated... When resizing arrays we chose dimensions to be multiples of 32.

3.3.3 Introducing constant memory and 3D indexing

Firstly it was tested for naive algorithm (with no array resizing and simpliest kernel). Kernels from file periodicCoulomb/vcoulomb_real.cu
Best results for Coulomb's convolution kernels (lattice 256x256x256, GeForce GTX 980M):

```
Calls    Avg        Min        Max       Name
1   4.1189ms   4.1189ms   4.1189ms   kernel_coulomb_3Didx0
1   3.7549ms   3.7549ms   3.7549ms   kernel_coulomb_3Didx1
1   3.2651ms   3.2651ms   3.2651ms   kernel_coulomb_3Didx_cnst  <- const mem
1   3.1699ms   3.1699ms   3.1699ms   kernel_coulomb_real0       <- 1D indexing
1   2.6150ms   2.6150ms   2.6150ms   kernel_coulomb_real_cnst   <- 1D indexing,
                                                                   const mem
```

This short overview suggest that the easiest way to optimize the kernel is to use constant memory and 1D indexing of array also in case of non-naive Coulomb implementation.

3.3.4 Spherical Cutoff Coulomb kernel optimization

Results of development versions of kernels (lattice 64x48x32 / 192x192x192, GeForce GTX 980M):

```
Calls   Avg [ms]   Min [ms]   Max [ms]   Name
100     3.392995   3.355504   3.743688   kernel_coulomb_sph_cutoff0<int=192, int=192, int=192>
100     3.308133   3.294319   3.591976   kernel_coulomb_sph_cutoff1<int=192, int=192, int=192>
100     3.329621   3.315282   3.616202   kernel_coulomb_sph_cutoff2<int=192, int=192, int=192>
100     2.927677   2.907552   3.176187   kernel_coulomb_sph_cutoff_cnst0<int=192, int=192, int=192>
100     2.855034   2.845691   3.210482   kernel_coulomb_sph_cutoff_cnst1<int=192, int=192, int=192>
100     2.850524   2.830299   3.095796   kernel_coulomb_sph_cutoff_cnst2<int=192, int=192, int=192>
```

Further optimization could be probably done by strided or batched read from global memory to kernels (and unrolling loops inside kernel_coulomb_sph_cutoff_cnst1 and kernel_coulomb_sph_cutoff_cnst2);

3.4 Callbacks

3.4.1 First tests

Results for code in file cufft_callback.cu (GeForce GTX980M, compute capability SM5.2):

Time(%)	Time	Calls	Avg	Min	Max	Name
73.37%	120.58ms	100	1.2058ms	1.1810ms	2.4289ms	ConvolveAndStoreTransposedC_Optimized
22.57%	37.097ms	100	370.97us	368.44us	377.17us	void spVector0512C::kernelMemCallback
4.01%	6.5895ms	101	65.242us	59.934us	472.30us	<callback_store_R2C>

Results for code in file cufft_no_callback.cu (GeForce GTX980M, compute capability SM5.2):

Time(%)	Time	Calls	Avg	Min	Max	Name
50.21%	22.094ms	100	220.94us	219.26us	223.16us	ConvertInputR
19.50%	8.5785ms	100	85.785us	70.814us	1.2094ms	ConvolveAndStoreTransposedC_Optimized
16.26%	7.1535ms	101	70.826us	66.974us	74.461us	void spVector0512C::kernelTex
14.03%	6.1728ms	101	61.116us	59.485us	63.742us	<callback_store_R2C>

This quick comparison show that FFT with callback load (void spVector0512C::kernelMemCallback + callback_store_R2C) takes about 43.62 ms and FFT without callback, but with preprocessing input in separete kernel (void spVector0512C::kernelTex + callback_store_R2C + ConvertInputR) takes about 35.28 ms, so is 20% faster. The issue is that a device function called from a kernel cannot use more registers than have been allocated to that kernel at launch time.
Moreover kernel ConvolveAndStoreTransposedC_Optimized became almost 15 times slover. We cannot find out what is the possible reason of such a behaviour. Maybe it is due to inconsistency in compute architectures of cufft_static library and the exploited machine?
Nevertheless the conclusion is that cuFFT kernels utilizing custom callback are less efficient and they probably will not accelerate program.
Same results on Tesla K80 with CUDA 7.5.

3.4.2 Test for naive Coulomb kernel

Callback load in inverse FFT:

Time(%)	Time	Calls	Avg	Min	Max	Name
%	ms		ms	ms	ms	
45.55	106.6498	1	106.6498	106.6498	106.6498	void dpRadix0256B::kernel1MemCallback
18.04	42.23892	3	14.07964	2.91e−03	21.20170	[CUDA memcpy DtoH]
11.54	27.01562	2	13.50781	13.41062	13.60501	void dpRadix0256C::kernel1Mem
9.26	21.68575	5	4.337150	9.28e−04	21.68182	[CUDA memcpy HtoD]
6.08	14.24234	1	14.24234	14.24234	14.24234	void dpVector0256C::kernelMem
5.60	13.10694	1	13.10694	13.10694	13.10694	void dpRadix0256C::kernel1Mem
3.93	9.200936	1	9.200936	9.200936	9.200936	void dpVector0256C::kernelMem

No callbacks:

Time(%)	Time	Calls	Avg	Min	Max	Name
%	ms		ms	ms	ms	
28.10	42.22948	2	21.11474	21.06722	21.16226	[CUDA memcpy DtoH]
18.70	28.10450	2	14.05225	13.57981	14.52468	void dpRadix0256C::kernel1Mem
18.70	28.09931	2	14.04966	13.23512	14.86419	void dpRadix0256C::kernel1Mem
14.38	21.61293	5	4.322586	8.96e−04	21.60922	[CUDA memcpy HtoD]
8.13	12.21691	1	12.21691	12.21691	12.21691	void dpVector0256C::kernelMem
8.03	12.06882	1	12.06882	12.06882	12.06882	void dpVector0256C::kernelMem
3.95	5.931899	1	5.931899	5.931899	5.931899	__density_times_vcoulomb_k__

This also confirms conclusions from results is a section above.

3.4.3 Problems with registers for callbacks

The issue is that a device function called from a kernel cannot use more registers than have been allocated to that kernel at launch time.
Source: https://devtalk.nvidia.com/default/topic/904009/unavoidable-register-spilling-with-cufft-callbacks/.
Possibly the neat solution would be to keep the callbacks in a register agnostic ptx form and then jit them as soon as the register constraints of the calling kernel are known. But don't know how to that...
https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-understand-fat-binaries-jit-caching/

3.5 Reductions

This section is considered due to approach to integration described in the next section.
Different types of solutions where investigated. The implementation of reductions in directory *integration* (files *reductions.cuh, thread_fence_reductions.cuh, cub_utils.cuh*), performace test in *reduce_test.cu*.
Results on GeForce GTX 980M, array 128x128x128 :

```
// CUSTOM BLOCK REDUCE
 22.03%  16.454ms      200  82.271us  3.0390us  173.79us  __reduce_kernel__<unsigned int=512>
  0.34%  255.16us      100  2.5510us  2.5270us  2.6560us  __reduce_kernel__<unsigned int= 64>

// THRUST
 16.53%  12.341ms      100  123.41us  122.21us  129.47us  <thrust1>
  1.14%  854.13us      100  8.5410us  8.2880us  8.9920us  <thrust2>

// CUB LIBRARY
 16.43%  12.270ms      100  122.70us  120.86us  124.38us  cub::DeviceReduceKernel
  0.50%  373.15us      100  3.7310us  3.4240us  4.1280us  cub::DeviceReduceSingleTileKernel
  0.28%  206.78us      100  2.0670us  1.9840us  3.2640us  cub::FillAndResetDrainKernel
```

Simple comparison between known methods of reductions reveals superiority of libraries over custom implementation.

3.6 Integrals (rectangles method)

As a supplement for the project we present methods for pefroming integrals of type:

$$\int \psi^{*}\left(\boldsymbol{r}\right) V\left(\boldsymbol{r}\right) \psi\left(\boldsymbol{r}\right) d^{3}r \approx \Delta x \Delta y \Delta z \sum_{ix,iy,iz} \left(\Re e\psi\left[ix,iy,iz\right]\right)^{2} + \left(Im\psi\left[ix,iy,iz\right]\right)^{2} \cdot V\left[ix,iy,iz\right]$$

Where assuming ψ and V are arrays of same size and cuDoubleComplex and double types respectively.
In quantum mechanics such integrals represent expectation values of operators.
Three types of solutions where investigated: based on thrust::inner_product, utilizing cuBLAS and self-written kernels. The implementation of intergals in directory *integration* (files *Integration.hpp, reductions.cuh*), performace test in reduce_test.cu.
Results on GeForce GTX 980M, array 128x128x128 :

```
// CUBLAS
 25.04%  60.297ms      100  602.97us  597.62us  624.95us  kernel_RC_mult
 19.45%  46.826ms      100  468.26us  463.54us  488.21us  dot_kernel
  0.16%  389.63us      100  3.8960us  3.7760us  4.9920us  reduce_1Block_kernel

// THRUST
 17.35%  41.783ms      100  417.83us  404.63us  435.00us   <thrust1>
  0.37%  889.25us      100  8.8920us  8.5440us  9.7590us   <thrust2>

// CUSTOM BLOCK REDUCE
 14.56%  35.055ms      100  350.55us  348.63us  355.51us  kernel_DZdreduce <unsigned int=512>
  0.14%  339.00us      100  3.3890us  3.2640us  3.5520us  __reduce_kernel__<unsigned int=512>
  0.11%  263.58us      100  2.6350us  2.5920us  2.7840us  __reduce_kernel__<unsigned int= 64>

// MEMORY TRANSFERS
  1.70%  4.0985ms        3  1.3662ms  1.2800us  2.7401ms  [CUDA memcpy HtoD]
  0.25%  590.57us      400  1.4760us  1.3430us  7.2950us  [CUDA memcpy DtoH]
```

Conclusions:

- Even without further optimization of kernel_RC_mult we must reject solution based on cuBLAS (dot_kernel is slow).
- Solution utilizing custom block reduce is suprisingly fast and beats Thrust by almost 20%.
- thrust::inner_product is almost 3.5x slower than simple reductions
- No solution with CUB Library was tested, but according to results of reductions methods tests it can also provide some speedup (can be faster than Thrust).

Appendix

Coulomb integral

$$\mathcal{F}\left[\frac{1}{4\pi}\frac{q^2}{|\boldsymbol{r}|}\right] = \frac{q^2}{4\pi}\lim_{\alpha\to 0}\int_0^{2\pi} d\phi \int_0^{\infty} dr \int_{-1}^{1} e^{ikr\cos\theta} e^{-\alpha r}\frac{1}{r}r^2 d\left(cos\theta\right) = \lim_{\alpha\to 0}\frac{q^2}{2ik}\int_0^{\infty}\left[\frac{1}{r}e^{ikrx}\right]_{x=-1}^{x=\ 1} e^{-\alpha r} r dr = \lim_{\alpha\to 0}\frac{q^2}{2ik}\int_0^{\infty}\left[e^{(ik-\alpha)r} -\right.$$

$$= \lim_{\alpha\to 0}\frac{q^2}{2ik}\left[\frac{1}{(ik-\alpha)}e^{(ik-\alpha)r} + \frac{1}{(ik+\alpha)}e^{-(ik+\alpha)r}\right]_{r=0}^{r=\infty} = \lim_{\alpha\to 0}\frac{q^2}{2ik}\left[\frac{(ik-\alpha)\,e^{ikr} + (ik+\alpha)\,e^{-ikr}}{(ik-\alpha)\,(ik+\alpha)}|_{r=\infty} - \frac{(ik-\alpha)+(ik+\alpha)}{(ik-\alpha)\,(ik+\alpha)}\right] =$$

$$= \frac{q^2}{2ik}\left[\frac{ik\left(e^{ikr}+e^{-ikr}\right)}{-k^2}|_{r=\infty} - \lim_{\alpha\to 0}\frac{2ik}{-k^2-\alpha^2}\right] = \lim_{R\to\infty}\frac{q^2}{2}\frac{2\cos\left(kr\right)}{-k^2}|_{r=R} + \lim_{\alpha\to 0}\frac{q^2}{k^2+\alpha^2}$$

$$= \lim_{R\to\infty} q^2\frac{1-\cos\left(kR\right)}{k^2}$$

See also:
http://physics.stackexchange.com/questions/7462/fourier-transform-of-the-coulomb-potential