Hybrid CPU-GPU Parallelization of Tomographic Reconstruction Algorithms in Version 1.2 of the IRXCT Windows Application

D Narcis Trinca, Institute of Measurement Science, Bratislava

Abstract

In the last few years, several packages for reconstruction in X-ray computed tomography have been developed. This report concerns the IRXCT (Iterative Reconstruction for X-ray Computed Tomography) Windows application for reconstruction and visualization of tomography tasks. Comparing to other tomography applications, the IRXCT application is complementary as it allows the user to run every reconstruction by means of the programmed interface instead of a script that runs a set of commands. One important function available in the IRXCT application is the possibility to see in real-time the reconstruction (after each iteration). The application is very fast, at it is using only Windows API functions, without any libraries needed to be installed along with the application.

In this report, we describe the parallelization possibilities available in Version 1.2 of the IRXCT application.

Keywords: computed tomography, GPU, iterative reconstruction, load balancing, parallelization

1 Introduction

In this report, we are describing the parallelization possibilities available in Version 1.2 of the IRXCT Windows application [5, 1] for reconstruction and visualization in tomography [2, 3]. Besides the SbIR algorithm [4], the SART algorithm [6] is also included in IRXCT. Adding other iterative algorithms to the application is very easy, as it requires only the addition of a new function for each algorithm. The IRXCT application has been designed so that reconstructions and associated calculations could be done by using the programme interface; so it has been designed so that it is somehow complementary to other applications where the reconstructions and other calculations are usually programmed manually by writing a script that will run the commands.

Three examples of tomography software available for download and regular use are the following: Astra toolbox, CASToR and NiftyRec.

Astra toolbox is a MATLAB and Python toolbox [7, 8] of high-performance GPU primitives for 2D and 3D tomography. It supports 2D parallel and fan beam geometries, and 3D parallel and cone beam. All of them have highly flexible source/detector positioning. A large number of 2D and 3D algorithms are available, including FBP, SIRT, SART,
CGLS. The basic forward and backward projection operations are GPU-accelerated, and directly callable from MATLAB and Python to enable building new algorithms. The source code of the ASTRA Toolbox is available on GitHub.

CASToR is an open-source multi-platform project [9] for 4D emission (PET and SPECT) and transmission (CT) tomographic reconstruction. This platform is a scalable software providing both basic image reconstruction features for "standard" users and advanced tools for specialists in the reconstruction field, to develop, incorporate and assess their own methods in image reconstruction (such as specific projectors, optimization algorithms, dynamic data modeling, etc) through the implementation of new classes.

NiftyRec is a software for tomographic reconstruction [10], providing GPU accelerated reconstruction tools for emission and transmission computed tomography.

The IRXCT application has available a number of functions that are complementary to other tomography software like the three software packages described above. For example, one important function is the possibility to see in real-time, that is after each iteration, the current reconstruction. Next, after a brief review of IRXCT, we describe the parallelization possibilities available in Version 1.2.
2 Brief General Description Of Version 1.2 Of The IRXCT Application

The IRXCT application is written in Visual C++ 2015, using only Windows API functions. Applications that use only Windows API functions are in general very fast, and this was the main reason for choosing to write all the functionalities using only Windows API functions. It consists of

1. a toolbar positioned at the top of the client area;
2. three child windows where the sinogram, the reconstruction (the middle child window) and the original image (for the simulation mode only) are shown;
3. a child window at the bottom-left corner where a plot showing in real-time the execution time taken at each iteration can be observed;
4. a list view at the bottom where mean signal, mean noise, and SNR (signal-to-noise ratio) calculations are shown (such calculations can be done once the final reconstruction is shown);
5. a report window that, during the reconstruction informs the user with the iteration number that is currently under calculation (along with execution time taken by each iteration), and at the end of the reconstruction shows a report that informs about the time taken to obtain the reconstruction.

The menu of the application consists of seven submenus, and is described in report. At start, the application looks like in Figure 1.

3 Hybrid CPU-GPU Parallelization Available With Version 1.2

The IRXCT Windows application has been written in Visual C++ 2015 (with NVIDIA CUDA 9.0 installed), as a CUDA 9.0 Runtime application. If there are nd detectors and nv views, then the total nd * nv projections are stored in the vector iIn1 declared globally as

```
unsigned int *iIn1;
```

The reconstruction algorithm being run, say the SbIR algorithm, is run with nT threads with Ids 0,1,...,nT-1 (the nT threads running on CPU), plus the GPU card currently selected if the menu option "Use GPU Card" is checked (GPU selection can be done with the menu option "Select GPU Card ..." as in Figure 2); the nT threads are started by the main thread of the application, which first does some preliminary initializations. The work to be done is splitted equally among the threads and GPU, by splitting the iIn1 vector of projections. If the menu option "Use GPU Card" is checked when the algorithm is run, then WorkGPU is a global variable that represents the amount of work to be done by the GPU.

If nrays is the number of projections in iIn1, then
nrays - WorkGPU

is the number of projections to be dealt with by the nT threads, and WorkGPU is the number of projections to be dealt with by the GPU; WorkGPU is initialized before the first iteration using one of the following two possibilities shown to the user in a Dialog Box after the user presses the "run" button: (1) as selected by the user (but no more than er) or, in case load balancing is checked, (2) er / 2 (er being the maximum number of projections, out of the total nrays, that can be dealt with by the GPU, due to memory constraints of the GPU). The nrays - WorkGPU projections are split equally among the nT threads.

Before the work of the nT threads and the GPU start, thread with Id 0 does the necessary initializations and memory allocations (using the already selected WorkGPU number prepared for the first iteration) on the currently selected GPU card using the functions

    cudaMalloc, cudaMemcpy, cudaStreamCreate;

after the preliminary initializations and memory allocations are done, the NI iterations are started. At each iteration we have four steps:

1. each of the nT threads does its own work of the SbIR algorithm, by running with its own subset of the projections; RII is a global matrix (available to all nT threads) that represents the reconstruction obtained at the end of the previous iteration, and each thread has
its own RI2 matrix (locally defined, but globally available to all nT threads through the vector pRI of global pointers) that represents the work done by the thread at the current iteration; in addition, thread with Id 0, before doing its own work of the SbIR algorithm, starts the work on the GPU to be done at the current iteration by running the function

\[ \text{iterations}_{\text{SbIR}}; \]

in addition, thread with Id 0, after finishing its own work of the SbIR algorithm, waits for the work on the GPU to finish by using the function

\[ \text{WaitOnAddress} \]

and when the waiting is finished it copies the work that was done by the GPU at the current iteration into the vector pri[0] (pri[0] being global, available to all nT threads);

2. all nT threads synchronize with each other using the call

\[ \text{EnterSynchronizationBarrier}; \]

3. each of the nT threads, using all nT RI2 matrices, does its own work in putting together all the work stored in the nT RI2 matrices and the matrix pri[0], to obtain the reconstruction RI1 of the current iteration; in addition, if load balancing is used, thread with Id 0 calculates the new WorkGPU number to be used at the next iteration by using the time taken by the CPU and the time taken by the GPU at the current iteration;

4. all nT threads synchronize with each other using the call

\[ \text{EnterSynchronizationBarrier} \]

so that the new reconstruction RI1 of the current iteration is obtained at the end of the step 3, since it is needed before the work on the GPU is to be started when the next iteration starts; in addition, if load balancing is used, it means that at step 3 thread with Id 0 has calculated the new WorkGPU number and thus each of the nT threads calculates now the new subset of projections (the nrays - WorkGPU projections are splitted among the nT threads);

4 How The New WorkGPU Number Is Calculated In Case Load Balancing Is Used

In case load balancing is selected by the user before the NI iterations start, then at each iteration thread with Id 0 calculates at the end of the iteration the new WorkGPU number by using the execution times of the CPU and
of the GPU at the current iteration. If $t_{CPU}$ is the time taken by the $nT$ threads to do in parallel the work of the SbIR algorithm (step 1 out of the four steps) at the current iteration and $t_{GPU}$ is time taken by the GPU to do its part of the work at the current iteration, then the code that we propose for obtaining the new $Work_{GPU}$ number is

\[
\begin{align*}
C/C++ \text{ code:} \\
t_{CPU} &= (((\text{float})(e_{CPU} - b_{CPU})) / ((\text{float})\text{CLOCKS\_PER\_SEC})) \\
t_{GPU} &= (((\text{float})(e_{GPU} - b_{GPU})) / ((\text{float})\text{CLOCKS\_PER\_SEC})) \\
\text{float } x &= \frac{(((\text{float})nrays) / ((\text{float})(*Work_{GPU})) \times t_{GPU})}{t_{CPU}} \\
(*Work_{GPU}) &= ((1.0 / x) \times (\text{unsigned int})(((\text{float})nIn1)) / ((1.0/x) + ((\text{float})nT))) \\
(*Work_{GPU}) &= (*Work_{GPU} < er ? (*Work_{GPU} : er))
\end{align*}
\]

To see how this formula works, consider the following example. If $nT=3$ and load balancing is used, and $t_{CPU}=0.5$ and $t_{GPU}=0.1$ and $nIn1=262144$ and $nrays=65536$ (nrays=65536 for each thread), and $Work_{GPU}$ is 65536, then $x=0.2$, meaning that the GPU was five times faster when compared to the $nT$ threads during the current iteration. Thus, at the next iteration, the GPU needs more work, so we divide $nIn1$ to $(5.0 + 3.0)=8.0$ and obtain 32768, which we multiply with 5.0 and obtain 163840, so the new $Work_{GPU}$ number to be used at the next iteration is 163840. The remaining 262144-163840=98304 are splitted equally among the 3 threads at the next iteration, so $nrays$ will be 32768 for each thread at the next iteration.

5 Results

To see how the proposed formula for load balancing works, we have run in simulation mode with the image shown in Figure 3, on a Windows 8.1 64 bits system with 4-cores CPU, and GPU card of compute capability 5.0 with 2 GB of memory.

The algorithm run is SbIR, and the parameters are: $nd=2048$ detectors, $nv=128$ views covering the $2\pi$ circle, distance from source to center of rotation is 1024.0, distance from source to detectors’ line is 2048.0, for a fan-beam geometry, with calculations using line integrals. The image in Figure 3 is of size 512 by 512 pixels, and the reconstruction is also of size 512 by 512 pixels.

After the image and the parameters are selected, we choose $NI=512$ iterations, check the menu option ”Use GPU Card”, choose the menu option ”3 Thread(s)”, and press ”run”. After the sinogram is calculated and displayed in the left child window, the application shows a Dialog Box where the user selects the work to be done on the GPU, as shown in Figure 4.

As shown, the user has two possibilities: either choosing a number between 512 and 132359 (132359 being the maximum number of projections, out of the total 262144, that can be dealt with by the used GPU, due to memory constraints), or checking load balancing. There are 3 cases that we have run:
Figure 3: image representing a cross-section of a nozzle created by additive manufacturing

Figure 4: in case GPU is to be used, then before iterations start we need to select the work to be done on the GPU
1. since we have 3 threads running and the GPU, so 4 computing units in total, then we choose 65536 for option 1 in the Dialog Box shown in Figure 4; the remaining 262144-65536 are split equally among the 3 threads, so each thread has 65536 projections to work on. In this case, after 512 iterations, the execution time was 334 seconds.

2. in the second case, we choose option 2 (load balancing), and the execution time in this case was 261 seconds.

3. in the third case, we choose 132359 for option 1, so the maximum possible for the GPU, and the execution time was 806 seconds.

Thus, as shown by this first example, load balancing is in general the best option to be chosen when splitting the work between CPU and GPU.

In a second example, we use the same image and parameters as in the first example, the only difference being that use 4 threads instead of 3 (so, we choose the "4 Thread(s)" menu option instead of the "3 Thread(s)" menu option"); the results are 339 seconds in the first case (choosing 52428 at option 1 instead of 65536, since now we have 5 computing units), 234 seconds in the second case when load balancing is used, and 798 seconds in the third case when the GPU does the maximum possible work. Thus again, load balancing is the best option for splitting the work between CPU and GPU.

6 How The Time Taken By The GPU Is Calculated

Computing the time taken by each thread to run its own work of the algorithm is easy, since we just have to run the clock() function before and after the respective code. But, finding out the time taken by the GPU to run its own work of the algorithm might not be so easy.

We have used the following method to compute the time taken by the GPU at each iteration. When the thread with Id 0 calls the iterations_SbIR function to start the work on the GPU, it uses a stream which is followed by a callback called MyCallback that will be run on the CPU once the work on the GPU is finished.

C/C++ code:

```c
(*g_TargetValue) = 0;
bGPU = clock();
iterations_SbIR<<<GpuThreads / 64, dimBlock, 0, stream>>>(...);
cudaStreamAddCallback(stream, MyCallback, NULL, 0);
```

The callback that we propose is as follows:

C/C++ code:

```c
void CUDART_CB MyCallback(cudaStream_t stream, cudaError_t status, void *data)
{
eGPU = clock();
}
```
InterlockedIncrement(g_TargetValue);
WakeByAddressSingle(g_TargetValue);
}

This means that bGPU is the time right before the work on the GPU is started and eGPU is obtained when the work on the GPU is finished and the callback is then run. The WaitOnAddress function that the thread with Id 0 uses to wait for the work on the GPU to finish blocks the thread until WakeByAddressSingle is called in the callback (WaitOnAddress blocks the thread until WakeByAddressSingle signals that the value of the g_TargetValue variable is changed, in our code from 0 to 1), at which time eGPU is already obtained and then the thread with Id 0 can calculate, using bGPU and eGPU, the time taken by the GPU at the current iteration.

7 Possible Improvements

1. bGPU is obtained in the current code by calling the clock() function right before the function to be run on the GPU is called; however, from the time when the function to be run on the GPU is called (more exactly, it is placed in the GPU queue) until the time when its actual execution on the GPU is started, might be a substantial delay. Thus, it is worth seeking methods to obtain the clock() directly on the GPU, right at the beginning of the code to be run on the GPU; this might possibly be accomplished using the clock() function of the GPU, but it might not work as it is, since the clock rate on the GPU is possibly different than the clock rate of the CPU.

2. concerning the four steps of the code run at each iteration by the nT threads, it could possibly be improved by using a mutex as follows: when, at the beginning of an iteration the function to be run on the GPU is called, then it might not necessarily be thread with Id 0 to do this but the first thread that reaches this part of the code; also, when WaitOnAddress function is called, it again might not be necessarily the thread with Id 0 to do this but instead the first thread that reaches this part of the code, so that in case the code on the GPU is already finished and the callback also run then WaitOnAddress returns immediately. One can use a mutex to let only the first thread that reaches these parts of the code to run the respective parts of the code. There is a tradeoff here that needs to be analyzed, since using a mutex might increase the execution time instead of decreasing it due to the acquiring and releasing of the mutex.

References

[1] IRXCT, Windows application available at the Github software repository github.com/IRXCT


[9] CASToR, package available at www.castor-project.org (open-source multi-platform project for 4D emission (PET and SPECT) and transmission (CT) tomographic reconstruction)

[10] NiftyRec, package available at niftyrec.scienceontheweb.net/wordpress (with Matlab and Python interfaces)