

List Mode PET reconstruction

Balázs Kovács

Abstract

PET technology has an important role in modern medical diagnostics. With this process we can view snapshot of a given part of the body's metabolism which provides more information than examining the organ's anatomy. Processing list mode measurement data is a demanding task, to solve this problem we use GPU which provides the necessary parallel architecture. The list mode approach's main benefit is that we can assign TOF (Time Of Flight) and energy information to every annihilation event. In this work, we will show in detail how can we effectively utilize the GPU to reconstruct from list mode data.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: PET reconstruction

1. The PET reconstruction problem and the solution

In Positron Emission Tomography (PET) we have to determine intensity distribution of the positron-electron annihilation events¹⁰. During an annihilation event two photons are born which impact with the detectors. We collect the simultaneous photon incidents in detector pairs, also called Lines Of Responses or LORs. We get an annihilation event list which will be our algorithm's input data.

To solve the reconstruction problem we use the expectation maximization (EM²) algorithm which determines an iteration scheme. Based on this, our algorithm consists of two parts: the forward projection and the back projection.

The iterative optimization process alternates forward projection

$$\tilde{y}_L = \sum_{V=1}^{N_{\text{voxel}}} \mathbf{A}_{LV} x_V^{(n)}, \quad (1)$$

then back projection:

$$x_V^{(n+1)} = \frac{x_V^{(n)}}{\sum_{L=1}^{N_{\text{LOR}}} \mathbf{A}_{LV}} \cdot \sum_{L=1}^{N_{\text{LOR}}} \mathbf{A}_{LV} \frac{y_L}{\tilde{y}_L} \quad (2)$$

in each of the $n = 1, 2, \dots$ iteration step where \tilde{y}_L is the estimated value of the current event, y_L is the measured value of the current event (it is always 1 in our case because one event corresponds to one detector incident), x_V is the estimated intensity of a voxel and \mathbf{A}_{LV} is an element of the system matrix which describes the probability that there is a detector incident in the current LOR starting from the current voxel.

In forward- and back projection steps, matrix elements \mathbf{A}_{LV} need to be estimated on-the-fly since the matrix is too large to store it in the GPU memory⁵.

2. Our proposal in detail

2.1. GPU implementation issues

The reconstruction algorithm can easily be adapted to GPU because the problem is well parallelizable^{11,4,1}. We can parallelize the forward- and the back projection based on list mode events, thus each thread processes a part of the event list. These threads are independent except for writing out the result during the back projection. We solve this problem using atomic operations which ensure that parallel threads can write the same memory bank but the operation will be sequential. Because of this reason, we have to minimize the usage of atomic operations.

2.2. Optimization concepts

- The CUDA shared memory is significantly faster than the GPU's global memory, hence we must maximize the usage of it.
- Threads in one CUDA warp wait for each other, so for optimal results they should not have greatly different running time. To ensure approximately equal running time, we divide the problem into subproblems which are distributed among the threads.
- To exploit the GPU's cache mechanism, we have to make the global memory accesses coalesced.
- We should avoid branching because the GPU runs both if branches before choosing the right one.

- The constant data (the detectors' and the voxel array's properties) should be stored in the constant memory of the GPU.

Regarding the listed concepts we build the algorithm the following manner⁹:

1. We sort the list mode events into three groups according to their dominant direction which is a cartesian coordinate direction whose scalar product with the event's direction is the greatest. This method helps us to fraction the problem into smaller pieces and improves the synchronization of the threads.
2. The voxel array is too big to be stored in the shared memory, thus, the kernels process the voxel array slice by slice. These slices' direction is determined by the current event group's dominant direction. We process the three event groups sequentially. A slice is small enough to fit in the shared memory, which is much faster than the global memory. Each multiprocessor loads a slice from the voxel array (one thread loads some of the voxels) which acts as a local cache. The more we can parallelize the problem the better, but if one thread processes only one event, there would be unnecessarily many global memory reads for one thread. So we have to find how many events should be processed by one thread keeping in mind that all of the GPU's multiprocessors should be in action but one thread should compute the most possible events' contribution.
3. We use a Tube Of Response (TOR or projection tube) for every event to model the positron range effect. The event's contribution to the voxels in the TOR is weighted with a Gaussian function. Every thread runs a double for-loop for each slice-event pair and because the events are selected into groups the for-loop processes all of the voxels which the event's TOR contains if the for-loop length is $\sqrt{2}(T_w + S)$, where T_w is the projection tube's width and S is the width of the slice (see figure 1). The reason of this is the angle between the event's and the current slice's direction is equal or less than 45° . The double for-loop can reach voxels which are outside the projection tube, these voxels are ignored depending on their distance from the voxel.
4. Because the projection tubes can intersect we have to use atomic operations in the back projection to refresh the voxel's value. Fortunately the new Fermi architecture GPUs support atomic operations on floating point numbers.
5. We store the list mode event list in the global memory. The memory accesses are coalesced because threads access the events sequentially.
6. In the back projection the Denom sum only depends on the geometry, thus, we can compute and store this value for all the voxels before starting the reconstruction. So if some of the events' geometry are the same, we don't have to compute the Denom several times.

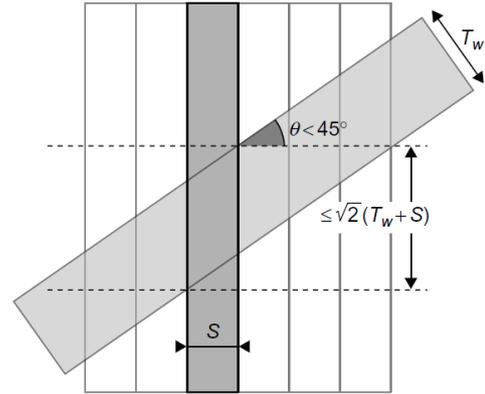


Figure 1: A slice and a event's projection tube's intersection

2.3. Implementation

2.3.1. Data structures

The method of the data storage is important regarding the algorithm's performance. We store all data in host and device memory because the kernels use the faster device memory, but we can only use the host memory to write out the computed results. We store the list mode events in a list and assign a structure to each event which contains the measured and computed value and the axial, tangential detector indexes which identify the event's geometry. Using these we can compute the event's detectors' physical coordinates inside the CUDA kernels. A list typically contains ten millions of events. Our GPU's memory can store approximately 50 million events because one event structure contains 6 ints and 2 floats which is 32B and the GPU's memory is $\approx 1.5GB$.

The voxel values are stored in 3D arrays. The Enum and Denom values of the voxel are stored separately because Denom values are computed before starting the reconstruction and Enum values are computed at every iteration. The size of the array depends on the resolution which is 96x96x96 in our case. The voxel arrays are not stored in the device memory because we don't need the device memory's speed since each kernel rarely writes the voxel arrays between big computational loads. Instead of the device memory we use a special CUDA memory called pinned memory⁸. The arrays reserve $volume_{width} \cdot volume_{height} \cdot volume_{depth} \cdot 2 \cdot 4B$ memory which is $\approx 7MB$ if $volume_{depth}$ is 96.

On the newest generation Fermi architecture GPUs the shared memory's maximal size is 48kB. Unfortunately we can't use a 128x128x128 voxel array because one slice would be $128 \cdot 128 \cdot 32bit \approx 65kB$. These slices can't be loaded in parts to the shared memory because if a TOR contains voxels of other parts then we have to access the global memory to load these voxels which is much slower than the

shared memory. Because of this we run the reconstruction algorithm on voxel arrays with maximum 96x96x96 size.

2.4. OSEM

Most of the list mode PET measurements contain approximately one billion events. Because of the reasons stated above this list does not fit in the GPU's global memory and we can use only a little part of the complete measured information, thus, the reconstructed image's quality isn't decent. We can use the OSEM (Ordered Subset Expectation Maximization) technique to solve this problem³. The main idea of this is that we make subsets from the full event list and we run the algorithm with other subset in every iteration. With this method we can use a lot more measurement information not using more device memory. The reconstruction converges faster and the result will be more accurate. Of course we load all of the subsets to the host memory.

2.5. Initialization

To ensure the double for-loop's constant length, we select the events depending on their direction into three groups before starting the reconstruction. We simply iterate through the list and put the X dominant events to the beginning of the list using swapping. We do the same with the Y dominant ones, of course we put these after the X dominant group. After this the Z dominant part will be at the end of the list. We sort in place and we don't make a new list because the event list can occupy huge amount of memory, so it may not fit in the memory one more time. We run the forward projection and the back projection separately for all the three groups.

2.5.1. Computing Denom

To compute the Denom value, we have to iterate through all LORs which are connected with the given voxel. We use only geometric information here, the LOR's measured and computed value is not important. Thus, the denom kernel don't process the event list but each thread computes one voxel's Denom value. The kernel searches all LORs which intersect with the given voxel so that it iterates through the given detector pairs and connects the current detectors' center points. To optimize this method we project the farther detector module's (secondary) corner points to the nearer detector module's (primary) plane, this intersection area determines the area of the intersection points of the primary detector module and the LORs which go through the voxel and the secondary detector module. Thus we have to iterate through only this area, not the whole primary detector module's surface. For each primary detector and given voxel we draw a line from the primary detector through the voxel and intersect with the secondary detector module, as a result we get a LOR whose contribution is added to the Denom.

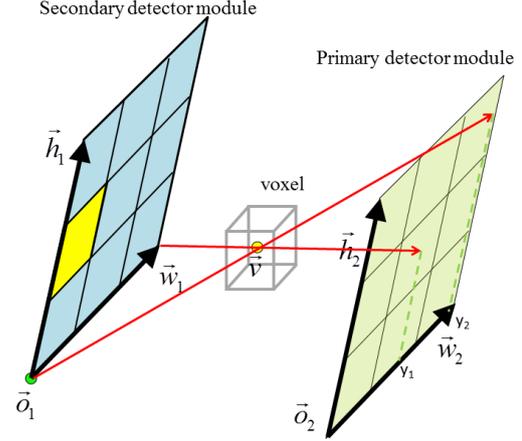


Figure 2: The secondary detector module's projection to the primary.

2.6. Forward projection

In the forward projection we compute an estimated value for every event in the list which is an intensity we would measure if we put the current voxel array inside the detector ring. We distribute the event list processing among the CUDA threads. This is a gathering algorithm, i. e., one thread computes an event's estimated value while it reads the memory several times but it writes only one memory bank which is not written by other parallel threads (see figure 3). Thus, we don't need to use atomic operations. Before calling the kernel, we initialize the events' estimated value with zero.

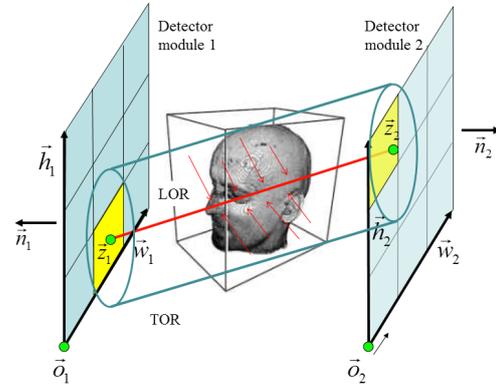


Figure 3: The forward projection for one event

The outline of the forward projection algorithm:

1. We choose the current slice and load it to the shared memory (we iterate through the slices with a for-loop).
2. We synchronize the threads in the block.
3. We pick the current event from the list.

4. Using the current event's detector coordinates we compute the event's physical position and intersection with the current slice.
5. We go through the voxels inside the intersection of the current event's projection tube and the current slice with a double for-loop.
6. We sum the contribution of these voxels. After the double for-loop we add this aggregate to the event's estimated value. We use a variable to store the aggregate, therefore we need only one global memory access when we set the event's new value.
7. We continue with the next event, start again from the 3. step.
8. We synchronize the threads in the block.
9. We continue with the next slice, start again from the 1. step.

Threads in one thread block uses one loaded slice together and every thread should load different values to the slice. Since the slice is two dimensional, a thread loads its part with a double for-loop so that it loads every $volume_width/block_width$ -th element in both for-loops (see figure 4). For example if $volume_width = 96$ and $block_width = 16$ then a thread loads every 6th element per dimension, so every 36th element.

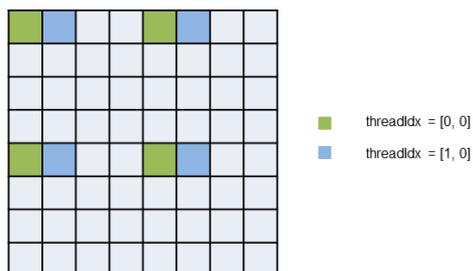


Figure 4: Loading one slice if $volumeDim = [8, 8, 8]$ and $blockDim = [4, 4]$.

2.7. Back projection - Computing variable Enum

In the back projection, similar to the forward projection, we process the event list, however, this is a scattering algorithm, therefore we have to use atomic operations. The reason of this is that an event contributes to all the voxels' Enum value which is inside the event's projection tube (see figure 5), so different parallel threads try to write to the same memory area. Before starting the back projection kernel we initialize the Enum voxel array with zeros.

The algorithm looks like the following:

1. We choose the current slice and initialize the shared memory with zeros (we iterate through the slices with a for-loop).

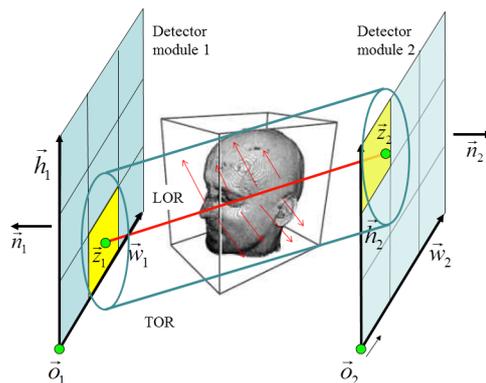


Figure 5: The back projection for one event

2. We synchronize the threads in the block.
3. We pick the current event from the list.
4. Using the current event's detector coordinates we compute the event's physical position and intersection with the current slice.
5. We go through the voxels inside the intersection of the current event's projection tube and the current slice with a double for-loop.
6. For these voxels we add the event's contribution to them.
7. We continue with the next event, start again from the 3. step.
8. We synchronize the threads in the block.
9. We add values stored in the shared memory to the Enum voxel array's corresponding voxels.
10. We synchronize the threads in the block.
11. We continue with the next slice, start again from the 1. step.

Like in the forward projection we use shared memory but we don't load the voxel array's values, but we add the slice's values, which is stored in the shared memory, to the voxel array's values. We can use the double for-loop and the indexing described in the forward projection (see figure 4).

Although the atomic operations are implemented in the hardware, we should avoid using them if possible. A decent solution is that every thread block starts to process a different slice, for example in every thread the slice's start index will be the thread block index. Therefore, when a slice's computation is finished and we start to add the slice's values to the Enum voxel array, less threads will try to write to the same global memory area because they write a part of a different slice per block.

In the back projection we have to use atomic add not only when we add to the Enum voxel array in the global memory, but when we increase the slice's values loaded in the shared memory. This time memory conflicts can occur between the block's threads. Since we use constant length for-loops to iterate through a given event's projection tube-slice inter-

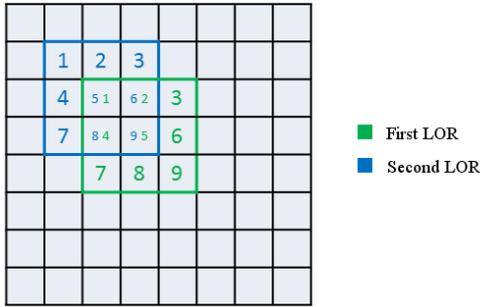


Figure 6: No memory conflict occurs if the two LORs intersect the slice in different voxels. The TOR width is 3 voxels and $\text{volumeDim} = [8, 8, 8]$. The numbers in the table show how the double for-loop iterates through the TOR-slice intersection.

section, therefore memory conflicts can only occur between two threads in the same block if their corresponding event and the slice's intersection is in the same voxel, i. e., their TOR-slice intersection is equal. In other cases there won't be memory conflict because the two double for-loop will write other memory areas in the same time (see figure 6). So, for a point source, where nearly all the events go through one voxel, there would be a lot of memory conflicts and the atomic operations would result in sequential execution.

After computing the Enum values a simple kernel corrects the voxel intensities according to the reconstruction algorithm:

$$x_V^{(n+1)} = x_V^{(n)} \cdot \frac{\text{Enum}}{\text{Denom}}.$$

3. Results

We ran our tests on an NVidia Tesla 2050 graphics card and we used the Derenzo phantom. The reconstructed voxel array size was $64 \times 72 \times 72$, the full event list consisted of 206 million events. This list is decomposed into five subsets which reserve 6.6 GB host memory. We compared our list mode approach to an existing binned reconstruction program ^{6, 12}.

So approx. 41 million events took part in one iteration (one OSEM group). The forward projection and the back projection times compared to the binned reconstruction:

Mope	Forward projection	Back projection
List mode	33 sec	120 sec
Binned	7 sec	11 sec

The binned reconstruction is evidently faster than the list mode algorithm since in the list mode case we process every event one by one.

The compared relative errors are the following:

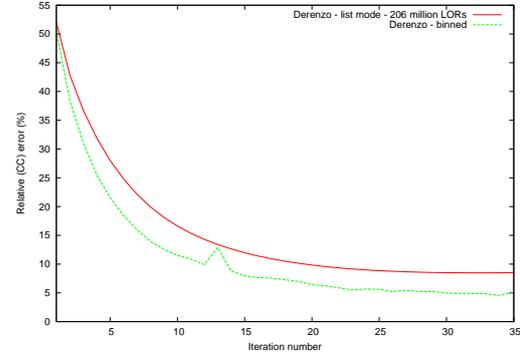


Figure 7: The relative (CC) error in function of the iteration number

The relative error is calculated based on difference between the reconstructed image and the provided phantom. We can see that the binned reconstruction converges faster and reaches lower error values. The reason of this can be that the binned reconstruction uses Total Variation ⁷. The reconstructed images are similar:

4. Conclusions

We can say that the GPU list mode approach has encouraging results but there are several possibilities to improve our algorithm. The most important is to exploit the list mode reconstruction's advantages using TOF and photon energy data which may decrease the running time and improve the quality of the reconstructed image.

5. Acknowledgements

This work is connected to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BME" project (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

References

1. Jakab G., Rác A., Bükki T., and Németh G. Fully GPU based real time corrections and reconstruction for cone beam micro CT. In *IEEE - High Performance Medical Imaging (HPMI): Science Symposium Conference Record (NSS/MIC)*, pages 4068–4071, 2009.
2. Stephen J Glick. Determination of the system matrix used in list-mode EM reconstruction of PET. *2007 IEEE Nuclear Science Symposium Conference Record*, (1):3855–3858, 2007.

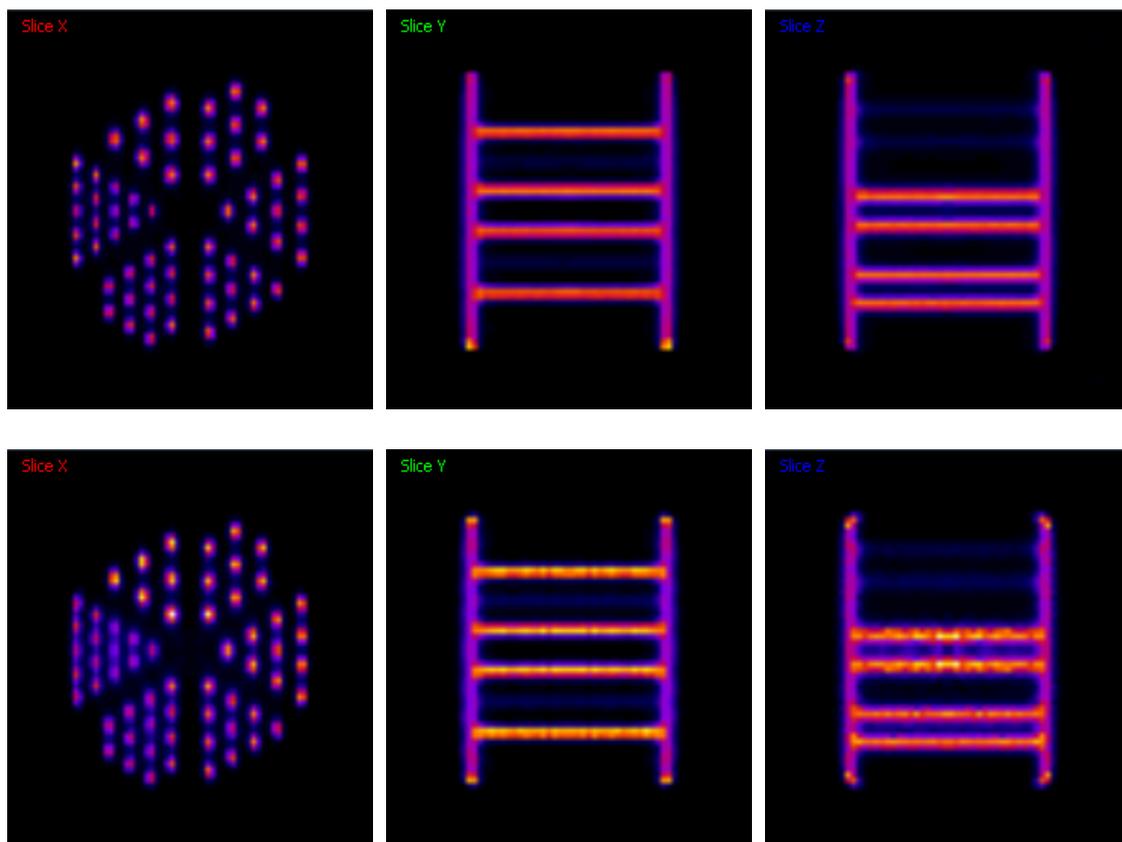


Figure 8: The binned (up) and the list mode (down) reconstruction's result, slices in the x , y and z direction.

3. Richard S. Larkin H. Malcolm Hudson. Accelerated Image Reconstruction using Ordered Subsets of Projection Data. *IEEE Transactions on Medical Imaging*, 13(4):601–609, 1994.
4. L. Szécsi L. Szirmay-Kalos. General purpose computing on graphics processing units. In A. Iványi, editor, *Algorithms of Informatics*, pages 1451–1495. MondArt Kiadó, Budapest, 2010. <http://sirkan.iit.bme.hu/~szirmay/gpgpu.pdf>.
5. M. Magdics, L. Szirmay-Kalos, B. Tóth, Á. Csenedesi, and A. Penzov. Scatter estimation for PET reconstruction. In *Proceedings of the 7th international conference on Numerical methods and applications, NMA'10*, pages 77–86, Berlin, Heidelberg, 2011. Springer-Verlag.
6. M. Magdics, B. Tóth, L. Szécsi, B. Csébfalvi, L. Szirmay-Kalos, A. Szlavecz, G. Hesz, B. Benyó, A. Cserkaszky, D. Légrády, Sz. Czifrus, A. Wirth, B. Kári, J. Lantos, D. Völgyes G. Patay, P. Major, G. Németh, T. Bükki, and B. Domonkos. Detector modeling techniques for pre-clinical 3D PET reconstruction on the GPU. In *Full 3D Concerence*, pages 375–8, 2011.
7. M. Magdics, B. Tóth, B. Kovács, and L. Szirmay-Kalos. Total Variation Regularization in PET Reconstruction. *Képfeldolgozók és Alakfelismerők VIII. Konferenciája*, pages 40–53, 2011.
8. Nvidia. The cuda's home page. <http://developer.nvidia.com/cuda/>.
9. Guillem Pratx, Jing-Yu Cui, Sven Prevrhal, and Craig S. Levin. *GPU Gems - Chapter 42 - 3-D Tomographic Image Reconstruction from Randomly Ordered Lines with CUDA*. Elsevier Inc., 2011.
10. Andrew J. Reader and Habib Zaidi. Advances in PET image reconstruction. *PET Clinics*, 2(2):173 – 190, 2007. PET Instrumentation and Quantification.
11. L. Szirmay-Kalos, L. Szécsi, and M. Sbert. *GPU-Based Techniques for Global Illumination Effects*. Morgan and Claypool Publishers, San Rafael, USA, 2008.
12. A. Wirth, A. Cserkaszky, B. Kári, D. Légrády, S. Fehér, S. Czifrus, and B. Domonkos. Implementation of 3D Monte Carlo PET reconstruction algorithm on GPU. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4106–4109, 24 2009-nov. 1 2009.