FPGA-Accelerated 3D Reconstruction Using Compressive Sensing

Jianwen Chen[§], Jason Cong[§], Ming Yan[†] and Yi Zou[§]
[§]Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA
[†]Department of Mathematics
University of California, Los Angeles
Los Angeles, CA 90095, USA
jianwen.chen@ieee.org,{cong@cs,yanm@math,zouyi@cs}.ucla.edu

ABSTRACT

The radiation dose associated with computerized tomography (CT) is significant. Optimization-based iterative reconstruction approaches, e.g., compressive sensing provide ways to reduce the radiation exposure, without sacrificing image quality. However, the computational requirement such algorithms is much higher than that of the conventional Filtered Back Projection (FBP) reconstruction algorithm. This paper describes an FPGA implementation of one important iterative kernel called EM, which is the major computation kernel of a recent EM+TV reconstruction algorithm. We show that a hybrid approach (CPU+GPU+FPGA) can deliver a better performance and energy efficiency than GPU-only solutions, providing 13X boost of throughput than a dual-core CPU implementation.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

The industry trend of CT imaging is moving towards low-dose CT. Although it is possible to reduce the dose directly and apply image-space denoising on the noisy FBP image, a more desired approach is to reduce the number of sampling used and apply compressive sensing-based iterative reconstructions. For a review of the CT image reconstruction and optimization-based iterative schemes, please refer to the recent survey [3].

This paper presents our effort to accelerate one iterative reconstruction algorithm called EM+TV [4], which extends classic Expectation Maximization (EM) [2] algorithm by introducing Total Variation(TV) regularization terms. We implemented the EM kernel completely on virtex 6 FPGAs. Our implementation is done at C-level by using AutoESL high-level-synthesis tool [1] from Xilinx.

FPGA'12, February 22–24, 2012, Monterey, California, USA.

2. ALGORITHM OVERVIEW

Typically, image reconstruction requires the number of samples (measurements or observations) that is above the Nyquist limits. By exploiting the sparsity of the objects, the number of samples can be reduced significantly. Compressive sensing technique exploits this fact to perform the reconstruction of signals or images. In our case, suppose the image is x, we make use of the sparsity of $|\nabla x|$ in the algorithm.

EM+TV reconstruction [4] tries to solve the non-linear optimization problem:

$$\min_x \int_\Omega |\nabla x| + \alpha \sum_{i=1}^{M} ((Ax)_i - b_i \log(Ax)_i)$$
$$x_j \geq 0, j = 1, \cdots, N \quad (1)$$

The first term is the TV term and the second one is the EM term. We ignore the mathematic details that can be referred in [4], but show the pseudo-code for the core computing functions instead.

2.1 Ray Tracing

EM algorithm is often implemented with a ray-driven forward-projection and a voxel-driven back-projection. To facilitate hardware sharing, we use ray-driven approach in both forward and backward projections. The code of the forward and backward projection is shown in Figure 1. The code first finds out the direction for the next voxel in the ray, then it performs multiply-and-accumulate operation to accumulate the sinogram or update the image. The tracing stops if the voxel hits the boundary of the object.

2.2 Intersection Computation

The $tracer\_precal()$ function is responsible for computing the intersection point of the ray with the object and find out the parameter required for the tracing. Given a source coordinate (s_x, s_y, s_z) and destination (d_x, d_y, d_z), the procedure finds out the intersection point with the object which is a cube $0 \leq x < N_x, 0 \leq y < N_y, 0 \leq z < N_z$. A number of divisions are used in the procedure.

3. IMPLEMENTATION & OPTIMIZATION

3.1 Parallel Backward Projection

The forward projection can be parallelized easily. A large number of parallel unit can operate on the forward ray tracers simultaneously for different source and detector pairs. For backward projection, there are dependencies among views. Moreover, even within one view, there are conflicts when two parallel units update one pixel. To resolve the data conflicts within one view, atomic functions that guarantee the mutual exclusion of an address in memory, can be used to handle such potential data conflicts. However, our target FPGA platform do not provide atomic operations

```
//EMupdate : ray-tracing algorithm
for all the views
for all the detectors
{
  tracer_precal(); // find initial ray parameters
  //  λx,λy,λz,λ0, vx,vy,vz,
  // Lenx,Leny,Lenz, signx, signy, signz
  if(mode==0) tempsino=0; //forward projection
  else  value= sinogram(..); //backward projection
  for (i=0;i<Nx+Ny+Nz;i++)//(tracer_loop)
     {
     if  (λx<=λy && λx<=λz) λ=λx;
     else if (λy<=λz)       λ=λy;
     else                   λ=λz;
      //Multiply accumulate (MAC) computation
     if(mode==0)            // forward projection
      tempsino+=imageData(vx,vy,vz)*(λ-λ0);
     else                   // backward projection
      imageData(vx,vy,vz)+=value*(λ-λ0);
     λ0=λ;
      //Find the next point on the ray
     if  (λx<=λy && λx<=λz){λx+=Lenx;vx+=signx;}
     else if (λy<=λz)    {λy+=Leny;vy+=signy;}
     else                {λz+=Lenz;vz+=signz;}
    //Exit conditions
     if(vx<0||vx>Nx-1) break;
     if(vy<0||vy>Ny-1) break;
     if(vz<0||vz>Nz-1) break;
     }
    if(mode==0) sinogram(..)=tempsino;
}
```

Figure 1: Ray Tracing Core Engine

on the memory system.[1] The only way to obtain a correct design is to enforce memory requests to complete sequentially. This has substantial overhead because the memory system is designed to be weakly ordered and supports parallel data access. We instead exploit algorithm-level changes to avoid the use of atomic operations. First, we ensure the computation for different views(sources) are done in a sequential fashion. For a same view, the detectors that are far enough are set to one group. Mathematically there will be no conflicts within the group and all tracers in one group can be processed in parallel. As illustrated in Figure 2, we can choose the tracer lines of the same pattern in one group.

3.2 Fixed Point Conversion

To reduce the area of our design, we convert floating point computation into fixed point. We use standard range analysis technique to obtain the range of all the values in our datapath. Because the algorithm is iterative, static precision analysis would generate quite pessimistic results. We use dynamic analysis instead to determine the number of fractional bits.

We try different number of fractional bits and compare with the floating point reference code. As illustrated in Figure 3, the bitwidth of the fractional part will influence the reconstruction quality greatly. When 18 bits (10^{-5}) are used, the fixed point version can achieve the same reconstruction quality of the floating point version. We enlarge the bidwidth by additional 2 bits to bring in more safe margins, and use 20 bits for the fractional part. Note that it is still possible to store all those array data using 32-bit data when we use 20-bit fractional part.

3.3 Streaming Architecture

Function $tracer\_precal$ and the tracer loop computation can be executed in a task-level pipeline. We synthesize the $tracer\_precal$ and the tracer loop individually to obtain their corresponding latency reports. Because the loop bound of the tracer loop is not known, we use an average loop bound from the simulation of the test data to compute the average-case latency of the tracer loop. The throughput of the memory interfaces is also considered. Roughly the latency of the $tracer\_precal$ is around $1/4$ of the latency of the tracer loop for a 128^3 test data. Because of this, we realize two $tracer\_precal$ modules and eight tracer loop module in a single FPGA. Each FPGA has 16 virtual memory channels, and each tracer loop module talks to two of them (one for read and one for write). The multi-FPGA system has 4 user FPGAs (Application Engine or AE), we distribute the work-load using SIMD fashion.

The diagram of our implementation in one FPGA is shown in Figure 4. To realize such a diagram in C level, we invoke the function $tracer\_precal$ twice and invoke the function of the tracer loop eight times. These different invocations take different FIFO channels and memory interfaces as parameters. The compiler can figure out that these function calls are independent and shall generate a parallel hardware.

The transform that converts the code in Figure 1 to a C code that calls two $tracer\_precal$ and eight $tracer\_loop$ seems counter-intuitive for software engineers. At higher-level, our manual step in this subsection can be viewed as a combination of loop unroll transform and loop distribution transform, where the distributed loops then take different unrolling factors. In practice, these decisions still need to be coded at a lower level.

The round robin distribution logic is also coded in the $tracer\_precal$ function. At the receiver side $tracer\_loop$, the control is just a simple counter to maintain the number of rays processed. Each $tracer\_loop$ would process a pre-determined number of rays. Note it is possible that the ray do not intersect with the object. In this case, the $tracer\_precal$ would send a special flag to denote that no processing is needed, but the counter should still be updated to obtain a correct exit condition.

The intersection computation we implemented is fairly generic. Currently, the control that sets the list of sources and detectors are also coded in the function, along with the lookup tables ROM for sin cos functions. Note it is very easy to change these controls to reflect another scanner machine setup.

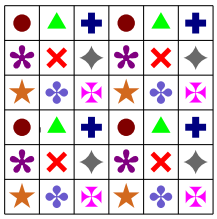

Figure 2: Ray Based Parallel Mapping

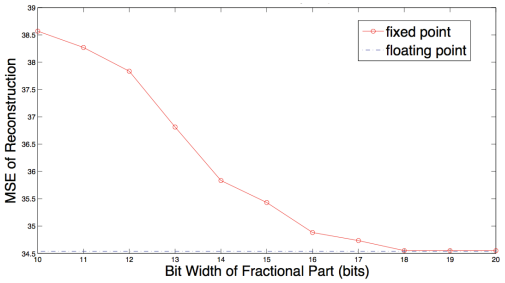

Figure 3: Fractional bit width and Reconstruction Quality

[1]It is possible to realize the atomic operation within the BRAM. The off-chip memory does not support atomic updates.

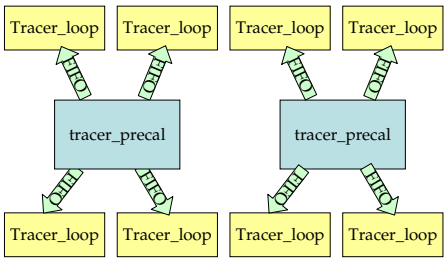

Figure 4: Overall Streaming Architecture Inside One FPGA AE

3.4 Prefetching

Generic HLS tools usually do not model board-specific IO systems. In our implementation, we model each memory access port with a request FIFO and a response FIFO. As shown in Figure 5, we need to invoke two parallel functions inside the hierarchy of $tracer\_loop$. One function is the "helper thread" $tracer\_loop\_addrGen$ which is responsible for sending memory requests for reads, and the other function is the "compute thread" $tracer\_loop\_compute$ which obtains data from response FIFO and write out the computed result into another request FIFO. This way, the helper threads can keep sending as many requests as possible (until the FIFO is full). Effectively, the helper thread is performing the prefetching of the required data, and the response FIFO serves as the prefetch buffer. Figure 5 depicts the architecture inside the $Tracer\_loop$ function.

3.5 Reducing the Data Accesses via Sparsity

The final output image of the compressive sensing algorithm is sparse. Also we know that the image voxel value is non-negative. Based on these two facts, we develop a simple heuristic to reduce the amount of data access. In the beginning of the iteration, we perform a single forward projection. If any accumulated sinogram value falls below a threshold, we conclude that any image value on that ray shall be close to zero. Based on this, we build a mask of the image called $image\_denote$. When we do the backward projection, we only update the voxels that are not masked. Note that this mask only need 1-bit data, so we merge this 1-bit data into the $imageData$ array. Through this way, we reduce the number of data access in the backward projection. Figure 6 shows the modified pseudo code.

3.6 Simultaneous Reconstruction of Two Images

After fixed point conversion, the external data accesses are all in 32-bit. The memory interface of our multi-FPGA platform supports 64-bit memory interface. Because of the data access in the tracing is somewhat random, it is hard to use the 64-bit interface to enlarge the application bandwidth. However, it is straightforward to use that to reconstruct two images simultaneously, by properly pack two 32-bit data from two images into a 64-bit data. These two images need to have exact machine setup where the $tracer\_precal$ part does not need to be changed.

We do not increase the number of MACs to support the 64-bit data. We measured that the external memory FIFO interfaces would return one data in about three cycles in the average case. [2] We simply enlarge the initiation interval (II) of the tracer loop from 1 to 2 to facilitate the sharing of MAC units.

[2]The peak rate is one data in every cycle. We did not reach such a high rate because our application logic is connected to a crossbar logic which performs arbitration and packet routing.

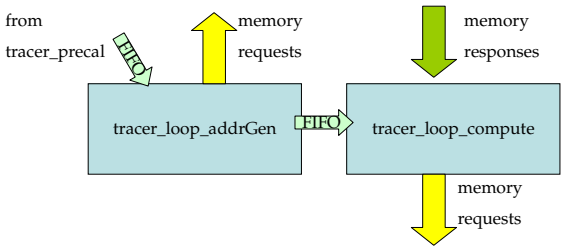

Figure 5: Streaming Architecture Inside One $Tracer\_loop$ Kernel

```
if(mode==0) // forward projection
        tempsino+ = imageData(v_x, v_y, v_z) * (λ − λ_0);
else // backward projection
    if(image_denote(v_x, v_y, v_z)==1)
        imageData(v_x, v_y, v_z)+ = value * (λ − λ_0);
```

Figure 6: Masking for Backward Projection

4. EXPERIMENTAL RESULTS

Our whole design is described in C and synthesized into verilog RTL using AutoESL HLS tool version 2011.1. The target hardware platform is Convey HC-1ex with 4 Virtex-6 LX760 user FPGAs. We designed the RTL interfaces for AutoESL tool to hook up with Convey's Personality Development Kit (PDK). Those interfaces are reused by a number of designs we implemented. PDK is the RTL-based synthesis and simulation environment for the HC-1ex platform. We synthesize the RTL generated by the AutoESL HLS tool along with the PDK infrastructure RTLs using Xilinx ISE 12.4.

Our test setup assumes a Cone-Beam CT system. Currently, we tested a phaeton data of size 128^3 which is supplied by authors of [4]. We have 36 views (sources) and the size of detector (destinations) is 301*257. According to [4], the EM+TV algorithm using 36 samples, can obtain a similar image quality that is obtained using FDK/FBP algorithm that requires 360 samples, resulting the radiation reduction by 10X.

4.1 Kernel Performance and Energy Consumption

Table 1 presents the performance and the energy consumption of the forward projection kernel and the backward projection kernel. The number is collected by averaging 1000 invocations. The performance on a dual-core CPU and many-core GPU is also reported. The CPU used is Intel Xeon 5138 with 2.13GHZ clock frequency and 35W TDP. The GPU1 column denotes Nvidia C1060 with 240 cores and 200W TDP. The GPU2 column denotes Nvidia GTX480 with 480 cores and 250W TDP. We parallelize the CPU code using OpenMP and implement the GPU kernel using Nvidia CUDA Toolkit 3.2. The throughput of the FPGA design is better than the latency because we can reconstruct two images simultaneously. The power of the FPGA application engine is measured by Xilinx xPower tool. We have 4 user FPGAs in the system. The actual system power of the Convey system is larger as the coprocessor memory, coprocessor PCB etc., also consume a lot of power.

From the Table 1 we can see that, when the latency of forward and backward is added together, our multi-FPGA engine is about 50% faster than the CUDA implementation on Tesla C1060, but about 2X slower than Fermi GTX480. When we consider the fact we can do two reconstructions simultaneously, that means our FPGA-engine is 3X faster than Tesla C1060 and in par with Fermi GTX480. The energy number is listed in the table as well. We can see that

Table 1: Performance and Energy Numbers for Computing Kernels for 128^3 data

	Power	Forward Projection		Backward Projection		Forward+Backward	
		Latency/Throughput(s)	Energy(J)	Latency/Throughput(s)	Energy(J)	Latency/Throughput(s)	Energy(J)
CPU	35W	1.81	63.4	1.67	58.4	3.48	121.8
FPGA	94W	0.305/0.153	28.7/14.4	0.308/0.154	29.0/14.5	0.613/0.307	57.7/28.9
GPU_1	200W	0.342	68.4	0.668	133.6	1.01	202
GPU_2	250W	0.085	21.3	0.276	69	0.361	90.3

Table 2: Area Results

	BRAM	DSP	LUT	FF	Slice
Consumed	79	68	113,355	104,099	36511
Total Available	720	864	474,240	948,480	118,560
Utilization	11%	7%	23%	10%	30%

Table 3: Application Performance and Energy Consumption

	Throughput(s)	Energy(J)
CPU	1189	41.6E3
GPU_1	361	72.2E3
GPU_2	114	28.5E3
Hybrid	92.0	12.7E3

the FPGA platform delivers a good performance with a much lower energy.

Note that it turns out that the execution time for backward projection is noticeably slower on GPU platforms. This is because the amount of data access is up-to 2X larger (we need to first read the voxel value and then write it back). Also we need to use more invocations (and synchronization) to avoid the conflicts and ensure the correctness. That also reduces the available parallelism. For the FPGA design, we use the same architecture for both forward and backward. Each PE is connected to two memory channels, one for read and one for write. Thus their execution times are similar. However, in the forward projection, the memory channel is somewhat under-utilized, because the number of writes is much smaller than reads. Potentially the forward projection can be made 2X faster if we separate the design for forward and backward.

Another interesting observation is that the Fermi GPU GTX480 is between 3 to 4X faster than Tesla C1060. The number of cores is 2X of C1060 and the peak off-chip bandwidth is about 1.6X (from 100GB/s to 160GB/s). So it is likely that there is an additional 2X performance benefit attributed from its cache systems. Our current FPGA design does not have a cache, but it is indeed worthwhile to investigate that possibility given the performance benefit we see from GPU.

The area results for the complete design are listed in Table 2. Note our core computing RTL consumes fewer logic slices, because the PDK infrastructure also consumes about 10% to 15% area. Most of the BRAM utilization is due to the PDK infrastructure.

4.2 Application Performance and Energy Consumption

We then test the application performance of the EM+TV algorithm on a hybrid configuration where the EM part is done by the FPGA-subsystem and the TV part is done by the GPU. In the application, the outer iteration iterates 100 times (the application calls 100 times of *EMupdate* and *TVupdate*), and the inner *EMupdate* step iterates 3 times (each *EMupdate* calls forward and backward projection routine 3 times).

Our hybrid configuration connects Fermi GTX480 onto the Convey HC1-ex platform. After one EM iteration completes, the image data is copied into the GPU memory space and the TV CUDA kernel starts. The data transfer would not add substantial overhead in this case. We measured that a pipelined data transfer (FPGA coprocessor-side memory to PCI-e) can reach close to 1GB/s. Each EM iteration only needs to copy 128^3 or 8MB image data to GPU. And similarly we need to do the transfer backwards when one TV invocation finishes. That only adds about 0.016s for each EM+TV iteration, or about 2s for the whole EM+TV application. Because the TV kernel is highly regular stencil computation, GPU is a good choice for that application kernel. The execution time of the TV is much shorter than EM. In the energy calculation for the hybrid configuration, we assume that GPU can be powered off when it is not actively running CUDA applications. In practice, a 10% to 15% idle power may remain.

Later, we also tested a phantom with size 256^3 and $512 * 512 * 256$, and obtained a similar speedup. The algorithm is roughly linear with number of voxels if the number of iterations are unchanged.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we present one FPGA-based implementation for ray-tracing EM kernels using AutoESL HLS tools. We further show that a hybrid approach provides good performance and potential energy savings. Currently, we are investigating different algorithmic or architectural approaches that can improve the data locality/reuse for the application.

6. ACKNOWLEDGEMENTS

This research is supported by the Center for Domain-Specific Computing (CDSC) which is funded by the NSF Expedition in Computing Award CCF-0926127.

7. REFERENCES

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE TCAD*, 30(4):473 –491, April 2011.

[2] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[3] X. Pan, E. Y. Sidky, and M. Vannier1. Why do commercial CT scanners still employ traditional, filtered back-projection for image reconstruction? *Inverse Probl*, 25(12), January 2009.

[4] M. Yan and L. A. Vese. Expectation maximization and total variation-based model for computed tomography reconstruction from undersampled data. In *Proc. SPIE Conference on Medical Imaging: Physics of Medical Imaging*, 2011.