SAND2013-6003C

Rethinking the Union of Computed Tomography Reconstruction and GPGPU Computing

Edward S. Jimenez[a] and Laurel J. Orr[b]

[a]Sandia National Laboratories, PO BOX 5800, Mail stop 0933, Albuquerque NM, USA;
[b]Sandia National Laboratories, PO BOX 5800, Mail stop 0927, Albuquerque NM, USA;

ABSTRACT

This work will present the utilization of the massively multi-threaded environment of graphics processors (GPUs) to improve the computation time needed to reconstruct large computed tomography (CT) datasets and the arising challenges for system implementation. Intelligent algorithm design for massively multi-threaded graphics processors differs greatly from traditional CPU algorithm design. Although a brute force port of a CPU algorithm to a GPU kernel may yield non-trivial performance gains, further measurable gains could be achieved by designing the algorithm with consideration given to the computing architecture. Previous work has shown that CT reconstruction on GPUs becomes an irregular problem for large datasets (10GB-4TB),[1] thus memory bandwidth at the host and device levels becomes a significant bottleneck for industrial CT applications. We present a set of GPU reconstruction kernels that utilize various GPU-specific optimizations and measure performance impact.

1. INTRODUCTION

Computed Tomography (CT) is an indirect 3D imaging technique in which an approximation of the 3D object is represented in a voxelized space and created from a set of 2D projection images (typically x-ray images).[2,3] The computational complexity required for volumetric reconstruction varies by the type of algorithm, but frequently ranges between $o(n^4)$ to $o(n^5)$ for single-pass and iterative algorithms.[4,5] There have been efforts in which the complexity has been reduced to $o(n^3\log(n))$[6,7] however; on a CPU-based system, this computation would still require an unreasonable amount of time to complete.

Graphics Processing Unit (GPU) technology has been applied successfully to various medical scale CT algorithms[5,8,9] (128^3 to 1024^3 voxels, 300 to 1000 projections). Yet, even with these improvements, the algorithm performance does not scale well when increasing the number of voxels or projections. Furthermore, adding more GPUs to a system does not mean scalable performance due to various hardware limitations. Previous work[1] has shown that large-scale CT is an irregular problem and hence has an unpredictable memory access pattern resulting in poor cache hit-rate among all GPU cache levels and types. This work will present a kernel that gradually evolves to exploit a graphics-specific architecture and measure incremental performance gains with respect to two large datasets.

2. CPU-BASED RECONSTRUCTION AND PORTING TO GPUS

The CUDA programming environment, as well as other GPU-programming languages (OpenCL, DirectCompute, etc.), have made GPGPU technology readily accessible to a large portion of the scientific computing community. Frequently, an honest first attempt to implement an algorithm on a GPU is to perform a brute force port of a CPU-based approach to a GPU-based implementation. It has been the experience of the authors that a blind CPU-to-GPU port of a properly parallelized algorithm will frequently yield a speedup in computation by a factor between 2× and 6×. This type of performance gain is typically sufficient for many small-scale applications and thus the added effort to exploit the GPU-specific hardware does not benefit the user significantly.

Further author information: (Send correspondence to E.S.J)
E.S.J.: E-mail: esjimen@sandia.gov, Telephone: 1 505 284 9690
L.J.O.: E-mail: ljorr@sandia.gov

A CPU-based reconstruction will typically loop over the projection data and iteratively update a single voxel on a given image plane. This process is repeated for every voxel on every image plane. For a multi-core CPU with n threads, the process is similar with the exception that n voxels are simultaneously updated. Currently, a typical system has an n that ranges from 2 to 32.

A simple GPU-ported kernel would allocate a computation thread for every voxel on an image plane and update each voxel in parallel with the given x-ray projection data iteratively. A CPU-based routine would loop over the image planes with a nested loop within iterating over projection data. Thus, for a reconstruction with N image planes and M projections, the kernel would be launched MN times.

Algorithm 1 CPU Kernel Launcher

Input: Projection Images $(P_1, P_2, \ldots, P_M)$, Scan Geometry (G), Voxels Per Image Plane (n)
Output: Voxelized Volume Reconstruction $I_1, I_2, \ldots, I_N$
  for Every image plane I_i **do**
    Allocate memory on GPU Device for I_i
    Initialize all values in GPU allocated array I_i to 0
    for Every Projection Image P_j **do**
      Allocate memory on GPU Device for P_j
      Upload P_j to GPU Device
      Launch Ported Kernel with n GPU threads (see algorithm 2)
      Free P_j on GPU Device
    end for
    Download I_i from GPU Device
    Free I_i on GPU Device
  end for

Algorithm 2 Ported Kernel

Input: I_i,P_j,G
Output: I_i updated with data from P_j
  Get thread id and designated voxel in I_i
  if Voxel in Region of Interest **then**
    Calculate back-projection path position, $\vec{b}$, within P_j
    Calculate bilinear interpolation weights, $\vec{w}$
    Calculate 2D interpolation on P_j based on $\vec{w}$ and $\vec{b}$
    Update designated voxel in I_i
  end if

Algorithms 1 and 2 are a possible implementation of a ported version of reconstruction. The largest performance gain is realized from performing the bilinear interpolation for every voxel in parallel. The most undesirable traits of the implementation is the large amount of data uploads, downloads, and kernel launches required. Each of these operations has significant overhead that sacrifices performance.[10]

3. EXPLOITING MASSIVE THREAD ENVIRONMENTS PROPERLY

Algorithms 1 and 2 are en example of a brute force implementation that requires minimal effort in porting over to a graphics processor. The next reasonable step would be to transplant the nested for-loop in algorithm 1 into the GPU kernel. Transferring the for-loops over to the kernel is desirable as this would reduce the number of kernel launches required to complete the reconstruction task as well as allowing the GPU to execute for longer periods of time which would improve the voxel processing throughput.

To implement the modification, one must still consider the almost arbitrary scan configurations and acquisition hardware. Thus, it is very likely that the entire projection dataset as well all imaging planes could not entirely reside on the device memory simultaneously. Therefore, a projection data and subset of image planes

will instead be used where the projection block contains a subset of the relevant projection images and the image plane block will contain the image planes that will be processed simultaneously. For large-scale reconstructions, this implementation will iterate over all image plane blocks and projection data blocks.

Algorithm 3 CPU Kernel Launcher (Block Scheme)

Input: Projection Images $(P_1, P_2, \ldots, P_M)$, Scan Geometry (G), Voxels Per Image Plane (n)
Output: Voxelized Volume Reconstruction $I_1, I_2, \ldots, I_N$
  Determine blocking of projection data $(B_1^P, B_2^P, \ldots, B_{M'}^P)$
  Determine blocking of image blocks $(B_1^I, B_2^I, \ldots, B_{N'}^I)$
  for Every image block B_i^I **do**
    Allocate memory on GPU Device for B_i^I
    Initialize all values in GPU allocated array B_i^I to 0
    for Every Projection image block B_j^P **do**
      Allocate memory on GPU Device for B_j^P
      Upload B_j^P to GPU Device
      Launch Blocked Kernel with n GPU threads (see algorithm 4)
      Free B_j^P on GPU Device
    end for
    Download B_i^I from GPU Device
    Free B_i^I on GPU Device
  end for

Algorithm 4 Blocked Kernel

Input: B_i^I,B_j^P,G
Output: B_i^I updated with data from B_j^P
  Get thread *id* and designated voxel in each image plane in B_i^I
  if Voxels in Region of Interest **then**
    for Each Image Plane I_h in B_i^I **do**
      for Each Projection image P_k in B_j^P **do**
        Calculate back-projection path position, $\vec{b}$, within P_k
        Calculate bilinear interpolation weights, $\vec{w}$
        Calculate 2D interpolation on P_k based on $\vec{w}$ and $\vec{b}$
        Update designated voxel in I_h
      end for
    end for
  end if

Algorithms 3 and 4 implement the blocking scheme discussed above, note that the CPU-based kernel launcher is essentially unchanged except for two-partitioning tasks and the nested for-loops now iterate over blocks of image planes (sub-volumes) and projection image blocks. Any blocking scheme can be used to accommodate any data format. The kernel presented in algorithm 4 will now take in blocked data, each of the n GPU threads launched will be assigned a subset of voxels to update with the projection data in B_j^P. Note that one could also launch more GPU threads on the GPU so that the outer for-loop in the kernel can be completely eliminated. We contend that for large-scale data, the performance difference is likely negligible as the management of the increased number of GPU-threads becomes burdensome and GPU memory bus would be over-saturated.

4. GPU HARDWARE INTERPOLATION

As mentioned earlier, the bilinear interpolation in the reconstruction algorithm is computationally expensive. One of the features of GPUs that differs from CPUs is hardware-based interpolation capabilities. The potential drawback is that for current technology, hardware-based interpolation is done in 24-bit precision[11] which differs

from the 32- or 64-bit precision that is frequently used. Fortunately, many imaging and inspection applications only utilize 16-bit precision. Thus, as long as a numerically stable approach is implemented, then precision will remain adequate.

To utilize the interpolation hardware, the projection data must be uploaded to the GPU as a read-only texture array. For current state-of-the-art, utilizing texture arrays allows for the kernel to utilize fast texture cache on the GPU for a potential boost in performance. To accommodate multiple projection images, one could utilize either multiple textures, a large texture with tiled projection images, or layered textures,[10,11] depending on the particular device being utilized.

Algorithm 5 CPU Kernel Launcher (HW Interpolation Scheme)

Input: Projection Images $(P_1, P_2, \ldots, P_M)$, Scan Geometry (G), Voxels Per Image Plane (n)
Output: Voxelized Volume Reconstruction $I_1, I_2, \ldots, I_N$
  Determine blocking of projection data $(B_1^P, B_2^P, \ldots, B_{M'}^P)$
  Determine blocking of image blocks $(B_1^I, B_2^I, \ldots, B_{N'}^I)$
  Allocate *texture* memory on GPU Device for largest $B_j^P \in \left\{B_1^P, B_2^P, \ldots, B_{M'}^P\right\}$
  for Every image block B_i^I **do**
    Allocate memory on GPU Device for B_i^I
    Initialize all values in GPU allocated array B_i^I to 0
    for Every Projection image block B_j^P **do**
      Upload B_j^P as a texture to GPU Device
      Launch Blocked Kernel with n GPU threads (see algorithm 6)
      Free texture B_j^P on GPU Device
    end for
    Download B_i^I from GPU Device
    Free B_i^I on GPU Device
  end for
  Free Texture memory allocated on Device

Algorithm 6 HW Interpolation Kernel

Input: B_i^I,G
Output: B_i^I updated with data from texture B_j^P
  Get thread *id* and designated voxel in each image plane in B_i^I
  if Voxels in Region of Interest **then**
    for Each Image Plane I_h in B_i^I **do**
      for Each Projection image P_k in B_j^P **do**
        Calculate back-projection path position, $\vec{b}$, within P_k
        Perform texture fetch from P_k at position $\vec{b}$
        Update designated voxel in I_h
      end for
    end for
  end if

Algorithms 5 and 6 implement algorithms 4 and 5 except they exploit the hardware interpolation capabilities of the GPU. Note that in algorithm 5, the projection data block is now allocated outside of the nested for-loop. The texture array is updated and reused during reconstruction. This allows for a reduced number of memory allocation and deallocations. Algorithm 6 has been simplified by eliminating the calculations of the weights $\vec{w}$ and the 32- or 64-bit interpolation operation and replacing it with a single texture fetch.

5. REGISTER AND GPU-CACHE OPTIMIZATION

Our final evolution of the reconstruction task is the implementation presented in Jimenez et. al.[1] This implementation involves improving algorithmic execution by minimizing wasted clock cycles on the device. To accomplish this, we exploit the following:

- GPU Memory Utilization
- Register Optimization
- Constant Memory
- Device Global Memory Fetches

Each will be addressed separately.

5.1 GPU Memory Utilization

Instead of uploading entire projection images, only relevant projection image data from a given projection block is uploaded. Determining relevant data is a trivial calculation and should have negligible effect on performance. The elimination of irrelevant projection data will improve the texture cache hit-rates as well as improve the utilization of device resources.

5.2 Register Optimization

When calculating $\vec{b}$, the order in which the calculations are executed as well as how many variables are utilized could potentially affect performance due to read-after-write dependencies and register pressure. Read-after-write dependencies have a latency of approximately 24 clock cycles for Nvidia GPUs.[10] Additionally, we implement a kernel such that register pressure is minimized as much as possible. This is achieved implicitly by the combination of optimizations in the list above.

5.3 Constant Memory

Until now, no mention of the parameters contained in $\vec{G}$ have been made. The vector $\vec{G}$ contains the scan geometry information (detector properties, source-to-detector distance, source-to-object distance, etc.). Throughout the reconstruction, $\vec{G}$ is fixed and is used in the calculation of $\vec{b}$. Therefore, we propose moving $\vec{G}$ into constant memory, which is a very fast user configurable cache that is disjoint from the L1- and L2-cache; thus reducing the traffic that goes through both cache levels.

5.4 Device Global Memory Fetches

When updating voxel information, memory bandwidth can be preserved by only updating voxel information when necessary. During iteration through projection data, a register will be updated, after these iterations complete, one voxel update is performed. The improvements are two-fold, not only is device bandwidth preserved for texture fetches, but also allows the L2-cache to more effectively feed the texture-cache.

5.5 Final Evolution

Algorithms 7 and 8 show an optimized implementation of the reconstruction task. Note that the kernel now only require a single input due to the texture implementation of the projection data and the constant memory assignment of the scan geometry parameters. This implementation will reduce the number of wasted clock cycles by providing a memory access scheme that keeps each GPU multiprocessor occupied with work. The approach to avoid read-after-write dependencies will be highly variable in its implementation due to the different schemes for each particular reconstruction algorithm.

Algorithm 7 Final CPU Kernel Launcher Optimized

Input: Projection Images $(P_1, P_2, \ldots, P_M)$, Scan Geometry $(\vec{G})$, Voxels Per Image Plane (n)
Output: Voxelized Volume Reconstruction $I_1, I_2, \ldots, I_N$
  Determine blocking of projection data $(B_1^P, B_2^P, \ldots, B_{M'}^P)$
  Determine blocking of image blocks $(B_1^I, B_2^I, \ldots, B_{N'}^I)$
  Upload $\vec{G}$ to GPU Constant Memory
  for Every image block B_i^I **do**
    Allocate memory on GPU Device for B_i^I
    Initialize all values in GPU allocated array B_i^I to 0
    Allocate *texture* memory on GPU Device for largest *relevant* projection block needed
    for Every Projection image block B_j^P **do**
      Determine Relevant Projection Data in B_j^P, ${B'}_j^P$
      Upload ${B'}_j^P$ as a texture to GPU Device
      Launch Kernel with n GPU threads (see algorithm 8)
    end for
    Download B_i^I from GPU Device
    Free B_i^I on GPU Device
    Free texture on GPU Device
  end for
  Free constant memory

Algorithm 8 Final Kernel Optimized

Input: B_i^I
Output: B_i^I updated with data from texture B_j^P
  Get thread *id* and designated voxel in each image plane in B_i^I
  if Voxels in Region of Interest **then**
    for Each Image Plane I_h in B_i^I **do**
      Set register to zero
      for Each Projection image P_k in B_j^P **do**
        Calculate back-projection path position, $\vec{b}$, within P_k while minimizing read-after-write dependencies
        Perform texture fetch from P_k at position $\vec{b}$
        Update register with texture fetch data
      end for
      Update designated voxel in I_h with register value
    end for
  end if

6. EVALUATION

All kernel implementations were tested on a high-end single node workstation consisting of a Supermicro X9DRG-QF Motherboard, 512 GB DDR3 System memory, dual Intel Xeon E5-2687W octo-core processors clocked at 3.1 GHz with hyperthreading for a total of 32 CPU threads, and 2 Nvidia/Next-I/O Tesla S2090 Devices connected via 4 PCI-E 2.0 16x host interface cards. Each S2090 device consists of 4 Nvidia Tesla M2090 GPUs with 6GB of GDDR5 device memory and 16 multiprocessors each for a total of 2048 CUDA cores per S2090. For this work, only 1 M2090 GPU will be used to minimize performance influence from the host.

Each kernel's performance will be measured against two datasets; the first is a 4000^3 voxel volume reconstruction from 1800 16 mega-pixel (4000×4000) images, the second is a trillion voxel reconstruction from $10,000$ 100 mega-pixel $(10,000 \times 10,000)$ images. All GPU kernels were written in CUDA (Version 5.0), host code was written in C++ using Microsoft Visual Studio 2008. Performance metrics consist of minimum, maximum, and average kernel runtime with respect to image plane position, as well as kernel runtime variance with respect to image plane position. As this work solely focuses on individual kernel runtime performance, other operations such

as data transfers between host and device, and data transfers from host to storage media will not be measured.

7. RESULTS

Figure 1 shows the average, minimum, and maximum kernel runtimes for the 4000^3 for each implementation of the kernels. As expected, each evolutionary step of the kernel yields an incremental performance increase. Note that across all implementations, kernel runtime increases among the central image planes of the volume, this is due to more projection data being processed per kernel launch and is expected. In order to allow comparisons to the ported kernel implementation, all performance times were normalized with respect to the projection blocks used for the other three kernels. Figure 2 contains the variances of the kernel runtimes for each kernel. Notice that the fully optimized kernel (algorithm 8) is the only kernel with significantly lower variance, this implies that not only is algorithm 8 fast (figure 1), but also behaves more consistently across the entire reconstruction volume. Figure 3 illustrates the incremental performance increase of the fully optimized version compared to all other implementations. We see that the overall performance improvement compared to the first implementation was almost a factor of 3.

Figure 4 displays the kernel runtimes for the center 100 image planes for the trillion voxel reconstruction. Runtimes are longer due to the increased computational expense. The results are still consistent with those observed in figure 1. The variance behavior of the trillion voxel dataset (shown in figure 5) is somewhat similar to that displayed for the 4000^3 dataset in that the fully optimized version exhibits the most consistent performance. Finally, figure 6 shows a slightly better performance improvement for the fully optimized kernel compared to the other kernels. This is due to the increased computational complexity and opportunity for parallelization. Also, the trillion voxel reconstruction will stress the limits of the hardware thus making discrepancies in performance more prevalent.

8. CONCLUSION

We have shown that GPUs can achieve a nontrivial increase in performance by properly utilizing GPU threads, specialized GPU hardware, and exploiting the unique cache structure to the advantage of the algorithm. This work has the potential to impact GPU applications in Green Computing, smart algorithm design, and high-performance computing. Many applications are not realizing the full effect of GPGPU technology, as this work is an example of the relatively minimal effort needed to redesign an algorithm and achieve significant performance gains. Many problems are arising where smart algorithm design will have a significant impact on power efficiency, performance, and computational time. This is especially true for new and emerging non-CPU computing architectures.

9. ACKNOWLEDGEMENTS

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] Jimenez, E. S., Orr, L. J., and Thompson, K. R., "An Irregular Approach to Large-Scale Computed Tomography on Multiple Graphics Processors Improves Voxel Processing Throughput," in [*Workshop on Irregular Applications: Architectures and Algorithms*], *The International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov. 2012).

[2] Barrett, H. H. and Myers, K. J., [*Foundations of Image Science*], Wiley-Interscience (2004).

[3] Buzug, T. M., [*Computed Tomography: From Photon Statistics to Modern Cone-Beam CT*], Springer-Verlag (2008).

[4] Feldkamp, L., Davis, L., and Kress, J., "Practical cone-beam algorithm," *Journal of the Optical Society of America A* **1**(6), 612–619 (1984).

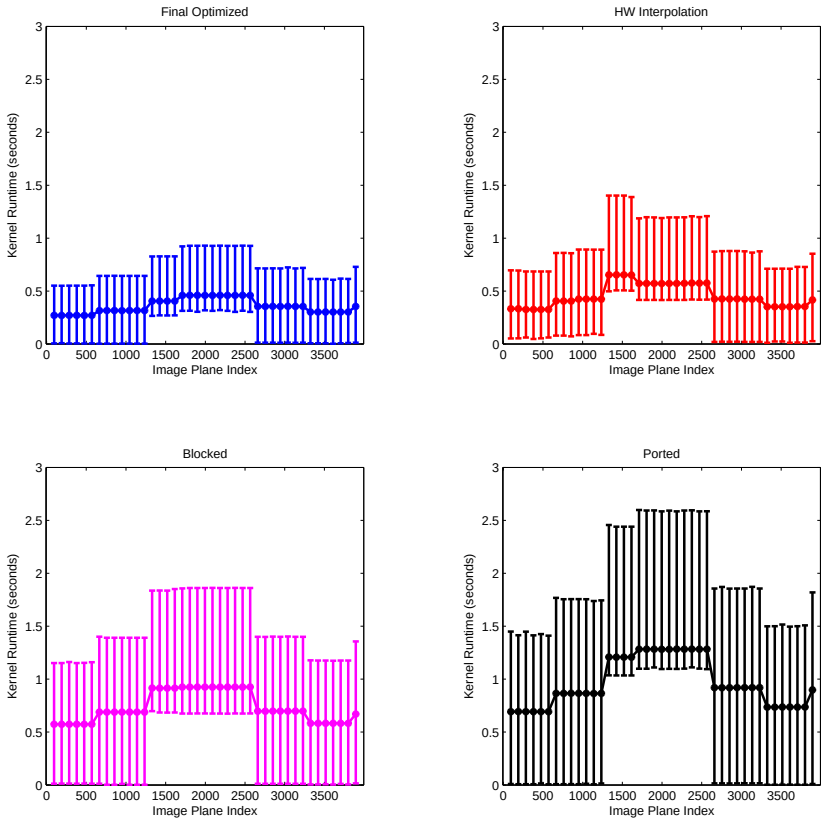

Figure 1. Average kernel runtimes with respect to image plane index for the 4000^3 voxel volume. Error bars represent the minimum and maximum runtimes

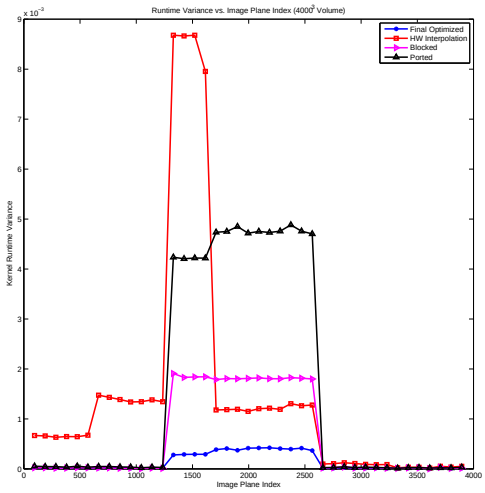

Figure 2. Average kernel runtimes variance with respect to image plane index for the 4000^3 voxel volume.

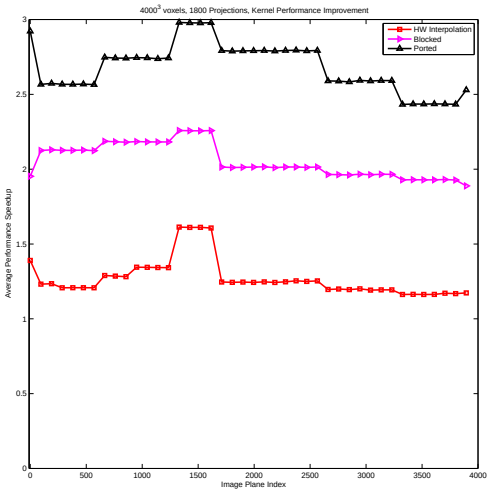

Figure 3. Average kernel performance improvement with respect to image plane index for the 4000^3 voxel volume.

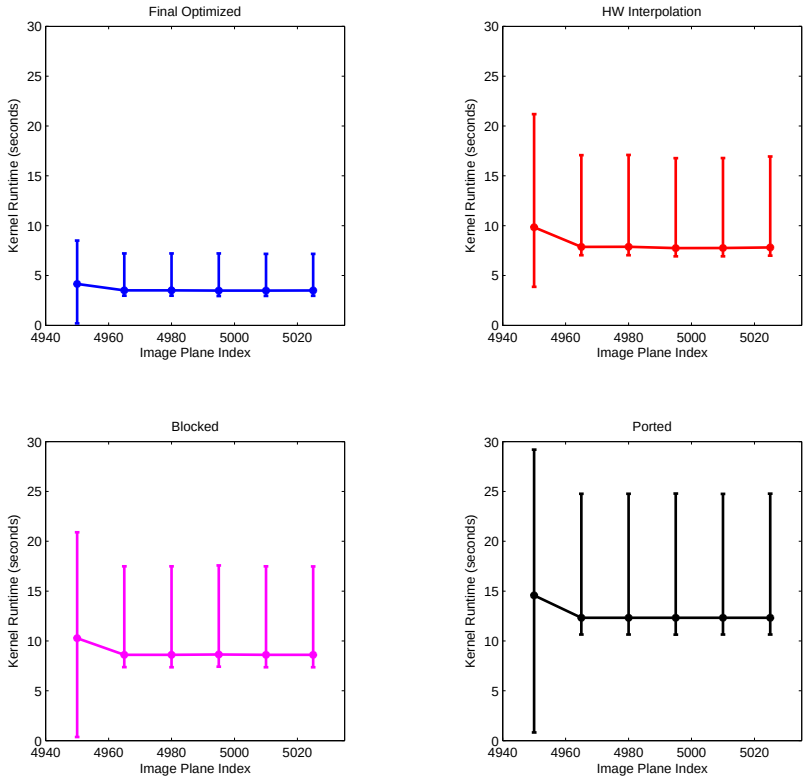

Figure 4. Average kernel runtimes with respect to image plane index for the center image planes of a 10000^3 voxel volume. Error bars represent the minimum and maximum runtimes

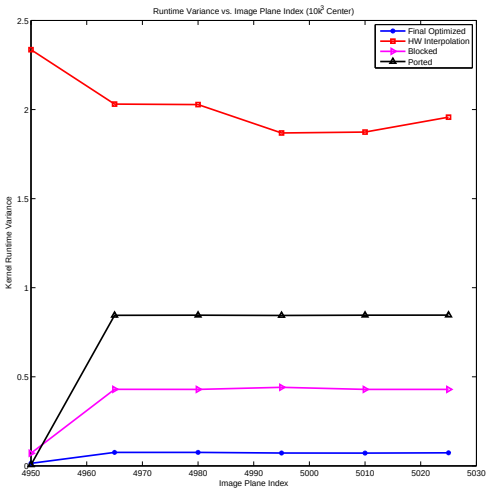

Figure 5. Average kernel runtimes variance with respect to image plane index for the Center image planes of a 10000^3 voxel volume.

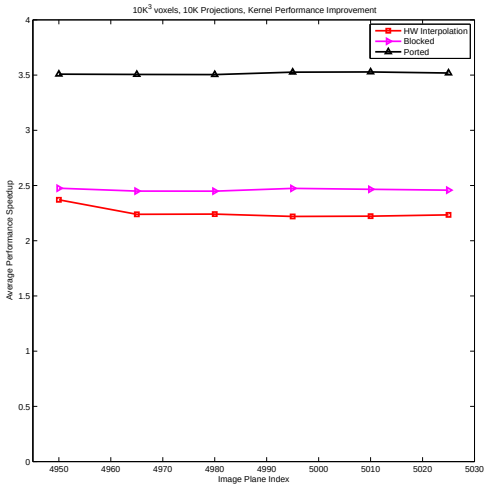

Figure 6. Average kernel performance improvement with respect to image plane index for the center image planes of a 10000^3 voxel volume.

[5] Xu, F. and Mueller, K., "Ultra-fast 3d filtered backprojection on commodity graphics hardware," in [*Biomedical Imaging: Nano to Macro, 2004. IEEE International Symposium on*], 571 – 574 Vol. 1 (april 2004).

[6] Xiao, S., Bresler, Y., and Munson, D.C., J., "Fast feldkamp algorithm for cone-beam computer tomography," in [*Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on*], **2**, II – 819–22 vol.3 (sept. 2003).

[7] Axelsson, C. and Danielsson, P., "Three-dimensional reconstruction from cone-beam data in o(n 3 logn) time," *Physics in Medicine and Biology* **39**(3), 477 (1994).

[8] mei W. Hwu, W., ed., [*GPU Computing Gems - Emerald Edition*], Morgan Kaufmann (2011).

[9] Xu, F. and Mueller, K., "Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware," *Nuclear Science, IEEE Transactions on* **52**, 654 – 663 (june 2005).

[10] Corporation, N., [*Nvidia CUDA C Best Practices Guide v5.0*], http://www.nvidia.com (2012).

[11] Corporation, N., [*CUDA C Programming Guide v5.0*], http://www.nvidia.com (2012).