Creating optimal code for GPU-accelerated CT reconstruction using ant colony optimization

Eric Papenhausen,[a] Ziyi Zheng,[b] and Klaus Mueller[c]
Visual Analytics and Imaging Lab, Center of Visual Computing, Computer Science Department, Stony Brook University, Stony Brook, New York 11794-4400

(Received 1 August 2012; revised 5 December 2012; accepted for publication 7 December 2012; published 28 February 2013)

Purpose: CT reconstruction algorithms implemented on the GPU are highly sensitive to their implementation details and the hardware they run on. Fine-tuning an implementation for optimal performance can be a time consuming task and require many updates when the hardware changes. There are some techniques that do automatic fine-tuning of GPU code. These techniques, however, are relatively narrow in their fine-tuning and are often based on heuristics which can be inaccurate. The goal of this paper is to present a framework that will automate the process of code optimization with maximum flexibility and produce a final result that is efficient and readable to the user.
Methods: The authors propose a method that is able to tune high level implementation details by using the ant colony optimization algorithm to find the optimal implementation in a relatively short amount of time. Our framework does this by taking as input, a file that describes a graph, such that a path through this graph represents a potential implementation. They then use the ant colony optimization algorithm to find the optimal path through this graph based on the execution time and the quality of the image.
Results: Two experimental studies are carried out. Using the presented framework, they optimize the performance of a GPU accelerated FDK backprojection implementation and a GPU accelerated separable footprint backprojection implementation. The authors demonstrate that the resulting optimal implementation can be different depending on the hardware specifications. They then compare the results of the framework produced with the results produced by manual optimization.
Conclusions: The framework they present is a useful tool for increasing programmer productivity and reducing the overhead of leveraging hardware specific resources. By performing an intelligent search, our framework produces a more efficient image reconstruction implementation in a shorter amount of time. [http://dx.doi.org/10.1118/1.4773045]

Key words: CT reconstruction, GPU, ant colony optimization, filtered backprojection, separable footprint

I. INTRODUCTION

Any CT reconstruction algorithm can be identified as a multiobjective optimization problem. The optimal result will provide the highest quality reconstruction in the shortest time. Many algorithms have been developed and extended, and good parameter settings have been identified to solve this problem under specific conditions.[1–3] However, if the boundary conditions change (i.e., noisier projections, different numbers of projections, stricter time constraint, GPU hardware, anatomy and pathology, etc.), the existing implementation is rendered suboptimal, and in some cases, useless.

In this paper, we use swarm optimization to determine an optimal CT reconstruction implementation for any given set of parameters. More specifically, we use the ant colony optimization algorithm to find an optimal implementation of a GPU accelerated FDK backprojection, described in Ref. 2 and a GPU accelerated separable footprint backprojection implementation.[4]

In this paper, we begin in Sec. II by discussing related work. Section III gives a brief description of the problem we are solving and the ant colony system optimization algorithm. Section IV gives a brief description of the graphics hardware used in our experiments and the structure of a CUDA program. Section V presents the details of our framework. Section VI presents the results of our experiments and Sec. VII concludes the paper.

II. RELATED WORK

Recent work has focused on finding good algorithmic parameters for iterative CT reconstruction.[5] Parameter tuning is critical in finding a good balance between image quality and reconstruction speed. The use of GPUs in accelerating CT reconstruction has also become very popular in decreasing reconstruction time.[6–8] However, not all GPUs are created equally; and there are many parameters to consider when creating a GPU accelerated program.

Fine-tuning GPU code can be a time consuming task. This typically requires making small changes to the program and observing the effect on performance. A number of tools have been developed to automate this tuning process.[9,10] These tools focus on finding a set of system parameters (e.g., memory layout, loop slicing, granularity, etc.) to achieve good performance. As a program becomes more and more complex,

however, it becomes increasingly difficult for these tools to capture the full space of potential parameter settings.

Whereas Xu and Mueller[5] focused on tuning algorithmic parameters and Klockner *et al.*[9] and Rudy *et al.*[10] focus on system level parameters, we set out to create a framework to allow the user maximum flexibility in exploring the full parameter space. By tuning system level parameters as well as high level algorithmic details, we increase the probability of finding the optimal solution for a given problem. Since this framework compiles and executes code to determine performance, the optimal implementation may change across different machines and this framework will be able to produce a machine dependent optimal implementation without direct programmer intervention.

III. ANT COLONY SYSTEM

To ease the reading of this paper, we provide a list of the symbols we use and their description in Table I.

The problem of finding an optimal implementation can be seen as a discrete optimization problem. We want to search through the space of all implementations that produce a specific output, to find the program with the smallest execution time. The cost function for this optimization problem can be seen in Eq. (1)

$$x^* = \text{argmin } E(x)$$
$$x \in \text{Q}. \tag{1}$$

Here, x^* is the optimal implementation. Q is the set of all implementations that produce a specific, desired output. The function E returns the execution time for implementation x.

Unfortunately, for a given problem, we do not know all the implementations that produce the desired output; so we rely on the user to provide an approximation to Q. The minimization function we attempt to solve in this paper is shown in Eq. (2)

$$x^* = \text{argmin } E(x)$$
$$x \in \text{Q}'. \tag{2}$$

Here, Q′ is the user provided approximation and is a subset of Q. It is described through a graphical representation in our framework. The degrees of freedom of Q′ are determined by the user, depending on what optimizations he describes and the problem he is trying to solve. Since we are only searching through a subset of the space of all possible solutions, the quality of the final solution is dependent on the user provided approximation. There is a chance that the globally optimal implementation lays outside of Q′ and so the success of this framework is largely dependent on the user's ability to narrow the search space in an effective way.

There are a number of optimization algorithms we considered before settling on ant colony optimization. Gradient descent is a popular method for solving optimization problems. Since the candidate solutions are non-numerical, however, it is unclear how one could calculate the gradient. Another optimization method we considered was genetic algorithms. This method of optimization, however, is susceptible to code bloat (i.e., large sections of code that do not contribute to the final output). It is also difficult to control the output of the candidate implementation with this approach. Candidate solutions can vary wildly, especially at the early stages of this approach, and there was the practical consideration that one solution could cause the computer to crash, at which point we would have to start the process over from the beginning. Ultimately, we decided that the ant colony optimization algorithm allowed us to have enough control over the candidate implementations to enforce the invariant that each candidate was correct in its outcome, while providing the flexibility for the framework to explore multiple solutions.

The ant colony system optimization algorithm is a part of the family of swarm optimization algorithms. It is a modification of the ant system algorithm, which was designed to mimic the way ants find the shortest path from the ant nest to a food source. Initially ants will choose paths randomly. Once an ant finds food, it will travel back to the nest and emit pheromones so other ants can follow that path to the food source. As other ants follow the pheromone trail, they emit pheromones as well, which reinforces the trail. After some time, however, the pheromone trail will evaporate. Given multiple paths to a food source, the pheromones on the shortest path will have the least amount of time to evaporate before being reinforced by another ant. Over time, the ants will converge to the shortest path.

The ant colony system was presented by Dorigo and Gambardella[11] and was applied to the traveling salesman problem. It modifies the ant system algorithm[12] in several ways to lead to a faster convergence rate. After an ant crosses an edge, the pheromone value of that edge is decayed according to Eq. (3)

$$\tau_{ij} = (1 - \varphi) \cdot \tau_{ij} + \varphi \cdot \tau_0. \tag{3}$$

Here, τ_{ij} denotes the pheromone quantity on the edge from state i to state j. The pheromone decay coefficient, φ,

TABLE I. Table showing the symbols that are used throughout the paper.

Symbol	Description
x*	The optimal implementation.
X	A candidate implementation.
Q	Set of all implementations that produce a specific output.
Q′	User provided subset of Q.
φ	Pheromone decay coefficient.
ρ	Evaporation rate.
p_{ij}^k	Probability that the edge from node i to node j is selected in the kth iteration.
τ_{ij}	Pheromone quantity on the edge from node i to node j.
$\Delta\tau_{ij}^{\text{best}}$	Inverse of the length of the edge from node i to node j if that edge is selected by the fastest ant.
τ_{ij}^{α}	Pheromone quantity on the edge from node i to node j; weighted by the constant, α.
η_{ij}^{β}	Predetermined desirability of the edge from node i to node j; weighted by the constant, β.

determines how much pheromone is decayed after an ant chooses the edge from i to j. The initial pheromone value, τ_0, is the value every edge has at the beginning of the program. Equation (3) reduces the probability of multiple ants choosing the same path.

After all ants have chosen a path, the pheromone of each edge is updated as follows:

$$\tau_{ij} = (1-\rho)\cdot\tau_{ij} + \rho\cdot\Delta\tau_{ij}^{\text{best}}. \quad (4)$$

The variable τ_{ij} has the same meaning as Eq. (3). The variable $\Delta\tau_{ij}^{\text{best}}$ evaluates to the inverse of the length of the best path if the edge from node i to node j was taken by the ant with the best path; otherwise it evaluates to zero. The variable ρ represents the evaporation rate. This leads to a pheromone increase on the edges taken by the ant that produced the best solution; while decaying the pheromones on all other edges. Equation (4) reinforces a path with pheromones proportional to the length of the path (i.e., execution time for our framework). This is particularly useful in our framework because certain optimizations are more effective than others, and this method of updating is able to capture that behavior. When transitioning from one state to another, the edge is selected probabilistically according to the following probability:

$$p_{ij}^{k} = \frac{(\tau_{ij}^{\alpha})(\eta_{ij}^{\beta})}{\sum(\tau_{ij}^{\alpha})(\eta_{ij}^{\beta})}. \quad (5)$$

Here, τ_{ij} determines the amount of pheromone on the edge from i to j, and η_{ij} defines some predetermined desirability of that edge (e.g., the inverse of the edge weight). The variables α and β are weighting factors for τ_{ij} and η_{ij}, respectively. The variable p_{ij}^{k} is the probability that an ant will select an edge that goes from state i to state j during the kth iteration.

IV. GRAPHICS HARDWARE

Modern GPUs follow a "single instruction multiple thread" (SIMT) model of parallel execution. In this model of execution, every thread executes the same instruction, but over different data. Since there will typically be more threads than processors on the GPU, threads must share GPU resources. With NVIDIA hardware, a group of 32 threads is organized into a warp. A group of warps is organized into a thread block, and a group of thread blocks are organized into a grid. This determines how many threads will be utilized during a Compute Unified Device Architecture (CUDA) kernel execution. Each thread in a warp executes simultaneously. Warps are scheduled onto the hardware in an efficient way. When one warp reaches a memory access or finishes executing, another warp is swapped in to take advantage of the vacant streaming multiprocessor. The implementation we attempt to optimize in our experiments use a C-like API called CUDA to program NVIDIA GPUs.

The GPU used in our experiments was the NVIDIA GeForce GTX 480. This graphics card contains 15 streaming multiprocessors. Each streaming multiprocessor contains 32 cores. Theoretical computing power of this graphics card is 1.3 TFLOPS. Like all NVIDIA graphics cards, this card has both on-chip and off-chip memory. Off-chip memory includes global, texture, and constant memory and typically incurs a latency of 400–600 clock cycles. On-chip memory includes shared memory as well as cache for texture and constant memory and is much faster than off-chip memory. The GTX 480 has a peak memory bandwidth of 177.4 GB/s for its 1.5 GB DDR5 device memory.

GPU accelerated applications have a large number of parameters that can be tuned for optimal performance. Occupancy (i.e., the ability to hide memory latency), the amount of work per thread, and memory bandwidth are all examples of the types of parameters that can have a large impact on performance. Tuning one parameter too much can often lead to a sudden decrease in performance in some other aspect of the application. This is what is known as a performance cliff.

V. IMPLEMENTATION

We use the ant colony system described in Sec. III to find and create an optimal implementation for a specific set of constraints. In order to do so, we define the structure of a program as a directed graph with a single source, at which every ant will start, and a single sink, where every ant will finish. The nodes of the graph correspond to source code snippets. A path from source to sink corresponds to a candidate implementation that can be compiled and executed. The output of the candidate implementation can then be measured and ranked among the other candidate implementations to find the ant with the shortest path for that iteration. The shortest path can be defined as a function of image quality and reconstruction time. By defining a graph in this manner, we have the added effect of introducing new implementations that were not previously defined.

The graph is constructed by creating a super source file. This super source file contains annotated sections of code. These annotations specify node id and incoming edges. Figures 1(a) and 1(b) show the graph and its corresponding super source file. Figures 1(c) and 1(d) show a potential path through the graph, and the corresponding candidate implementation. This super source file is then submitted as input to our program, which converts it to its graph representation and runs the ant colony system algorithm to produce an optimal implementation.

One aspect of our program differs from the traditional ant colony system algorithm. There is no predetermined desirability, η. There is no way of determining edge weight before running the algorithm. We can still apply the ant colony system algorithm by only considering the pheromone value, τ, when looking at an edge. This is equivalent to setting η to one, for all edges. Equation (6) shows how edges are selected by ants. Our experiments indicate that this still converges to an optimal solution

$$p_{ij}^{k}\frac{(\tau_{ij}^{\alpha})}{\sum(\tau_{ij}^{\alpha})}. \quad (6)$$

Since graphics hardware plays such a prominent role in CT reconstruction, our framework provides the option of expanding the graph provided in the super source file to

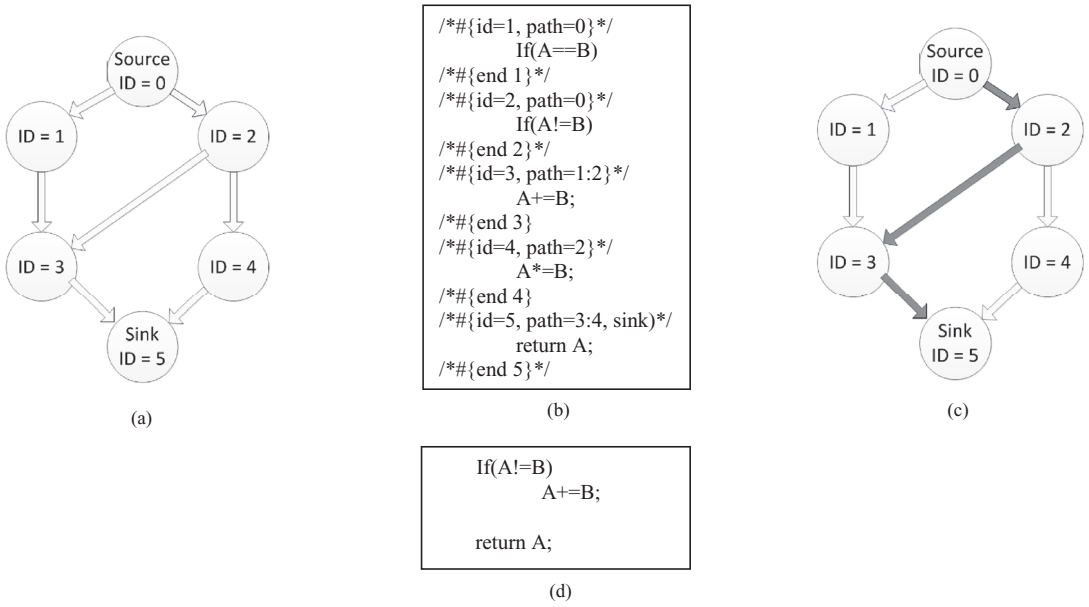

FIG. 1. An illustration of the framework presented in this paper. (a) A graph representing all possible implementations of a program. (b) The super source file represented by the graph in (a). (c) A path is selected through the graph. (d) Source code corresponding to the path selected in (c).

account for different grid and thread block sizes. This is done by copying the code snippets that contain the threads unique ID and offsetting the ID by the grid dimension. This allows the framework to implicitly increase the workload for each thread. Figure 2 shows an example of this. The grid and thread block dimensions determine the granularity of each thread. The smaller the grid and block size, the more work each thread will perform. In this specific example, a thread in Fig. 2(b) computes two final results and stores them into the respective target locations in memory, whereas, a thread in Fig. 2(a) only computes one result.

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
                    .
                <code>
                    .
F_L[tid] = result;
```

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
                    .
                <code>
                    .
F_L[tid] = result;
                    .
                <code>
                    .
F_L[(blockIdx.x + 8) * blockDim.x + threadIdx.x] = result;
```

FIG. 2. Sample code demonstrating how thread granularity can be increased implicitly. (a) Source code representing a thread granularity of one (e.g., grid = 16, thread block = 16). (b) Source code representing a thread granularity of two (e.g., grid = 8, thread block = 16).

VI. EXPERIMENT AND RESULTS

We used the framework presented in this paper to create an optimal GPU accelerated implementation of the FDK backprojection algorithm described in Ref. 2. This backprojection implementation is then tested with the help of the RabbitCT framework.[13] We also used this framework to create an optimal GPU accelerated implementation of the separable footprint backprojector.[4] Our implementation contains variations of the work of Wu and Fessler[14] and we demonstrate that this framework can produce different variations of this implementation for different hardware without direct programmer intervention.

VI.A. FDK backprojection

We chose to compose the graph out of the three major implementations presented in Ref. 6. An ant's path from the source to the sink represents either the first, second, or third configuration presented in Ref. 6 or some combination of the three. In this graph, we also added a fourth configuration in which two projections are loaded per kernel call. Certain paths can lead to implementations that produce bad images (i.e., images of low quality). Therefore, computation speed is not the only criteria for evaluating performance. If the reconstructed image has an error that is too high, the corresponding ant is given a bad score; thus reducing the probability of other ants selecting that path in future iterations. The error is measured using the mean squared error with a suitable reference image. For more information about the structure of this graph, see the supplementary material.[17]

We ran our framework with 30 ants for 5 iterations. Our super source file described a graph that contained 25 nodes.

TABLE II. Runtimes of ant optimized and hand optimized implementations.

Configuration	Volume	Time (s)
Ant optimized	256^3	2.54
Hand optimized (Ref. 6)	256^3	2.71
Ant optimized	512^3	6.07
Hand optimized (Ref. 6)	512^3	6.07

This graph, however, is replicated for 16 different grid and thread block dimensions; creating a graph that contains 400 nodes. Table II shows a comparison of the timings of the FDK implementations that were produced through our framework with the results presented in Ref. 6, which were obtained by manually optimizing the code. For the 256^3 implementation, we found a faster implementation. This configuration loads two projections per kernel invocation and has a thread granularity of two in the x direction. For the 512^3 implementation, our framework produced the same code as in Ref. 6 but determined this without the need for lengthy manual tuning. Figure 3 shows a slice of the reconstructed volume. The quality of the reconstruction for the implementations produced by this framework was the same as the quality produced in Ref. 6.

The runtime of our framework is dependent on the scale of the application it is trying to produce. For each ant, source code is generated, compiled, and executed. For the experiments that we ran, it took approximately 2 h for all 30 ants to complete 5 iterations. Although the increase in performance is very little, a lot of time was spent during the manual optimization. We spent approximately 2 days (i.e., 8–10 h each day) to reach this optimization through hand tuning. This shows that this framework is a powerful tool in increasing programmer productivity.

To reduce the runtime for the framework, this graph could have been pruned by eliminating nodes that correlate to configurations that we know have bad performance. In our experiments, we included the naïve configuration explained in Ref. 6 as a possible implementation. By pruning the graph of bad implementations, we could reduce the number of ants; thus, greatly reducing the amount of time required by our framework. We note that although it takes some time to find the right optimization parameters, the code produced in this manner can be reused for any new CT reconstruction task with the same boundary conditions. Therefore, the optimization overhead is well amortized.

FIG. 3. Slice of the FDK reconstructed image.

VI.B. Separable footprint implementation

Separable footprint is a technique developed in Ref. 4 for forward and backprojecting. It is similar to splatting and builds off of Ref. 15 by separating the transaxial and axial footprint in order to increase performance. The authors Long and Fessler of Ref. 4 have shown that the separable footprint technique is more accurate than the popular distance driven method.[16]

Similar to the FDK backprojection implementation, we have developed multiple solutions for the GPU accelerated separable footprint implementation. Though there are multiple solutions, the differences between them are relatively small and are mostly low level optimization details. In this section, we will describe the general idea behind the GPU implementation of the separable footprint backprojector described in Ref. 14.

The separable footprint backprojector involves calculating the axial, t, footprint and the transaxial, s, footprint for each voxel. This essentially determines how much each detector cell contributes to the voxel. This is more accurate than the FDK approach because there is no interpolation. Since the footprint of each voxel is independent from other voxels, we can parallelize this process.

In Ref. 14, the authors leverage CUDA and GPU hardware to accelerate the separable footprint backprojector. They separate the backprojector into multiple kernel calls. First, they calculate the transaxial footprint on the GPU. The next kernel then parallelizes over x, y, and t and computes the sum of the product of the transaxial footprint and the projection value for each s value. The final kernel call accumulates the product of the previous kernel with the axial footprint for each x, y, and z value. This process is then repeated for each projection.

VI.C. Separable footprint experiment

Our graph structure contains multiple variations of the same basic implementation that we described in Sec. VI.B. The differences are mainly in the types of memory used (i.e., texture, surface, global, etc.), and the thread granularity. This experiment differs from the FDK experiment in that we attempt to optimize multiple kernels simultaneously. The graph for this experiment contains over 20 000 possible implementations. An exhaustive search would take approximately two weeks to find the optimal solution; and so clearly this is impractical. A large cluster of GPUs would be required to perform an exhaustive search in a reasonable amount of time. Since the number of possible implementations grows exponentially with the number of kernels, however, the search space can quickly become too large for clusters of GPUs to search in a reasonable amount of time. This multikernel optimization is especially useful in determining the optimal usage of surface memory. Surface memory will often be written to in one kernel, and then read from in a different kernel. Since surface memory cannot reliably be written to and read from in the same kernel, using surface memory can restrict other implementation choices.

TABLE III. Runtimes of framework produced separable footprint implementations. The timings show how long it takes to backproject 364 projections that contain 1014 × 374 detector cells.

Hardware	Volume	Time (s)
GT 335 m	256^3	137.38
GTX 480	256^3	15.95
GT 335 m	512^3	N/A
GTX 480	512^3	72.19

Since surface memory is not supported on all graphics cards, some options may be closed off depending on the hardware. If it is not supported, ants that choose a path that attempts to use surface memory will create an implementation that cannot be compiled. Our framework, however, is robust to implementations that are not necessarily supported by the current hardware. Ants that choose this path will simply be assigned a bad score and the framework will converge to another solution.

In addition to the GTX 480, we also performed experiments on the NVIDIA GeForce GT 335 m GPU. This hardware has a compute capability of 1.2 and contains 78 CUDA Cores and 20 GB/s global memory bandwidth. By performing experiments on multiple GPUs, we show how the optimal implementation changes depending on the hardware.

We used this framework to optimize the separable footprint backprojector for a 256^3 volume and a 512^3 volume. On both the GT 335 m and the GTX 480, we found that the optimal implementation stores the projection data in texture memory and parallelizes in x, y, and z during the final kernel call. Since surface memory is not supported on the GT 335 m graphics card, there were some major differences in what each GPU found as the optimal implementation. On the GTX 480, our framework chose to use surface memory whenever it was an option. This was actually somewhat surprising. We expected that the overuse of surface memory would have led to a lower cache hit rate, especially during the final kernel, and would negatively impact performance. The results of this experiment can be seen in Table III. Timings for the 512^3 volume could not be obtained for the GT 335 m because the memory requirement was too large. For the GTX 480, the implementations our framework produced were the same for both the 256^3 and 512^3 volume. A slice from the backprojected image can be seen in Fig. 4. See the supplementary material for more information on the separable footprint backprojection experiment.[17]

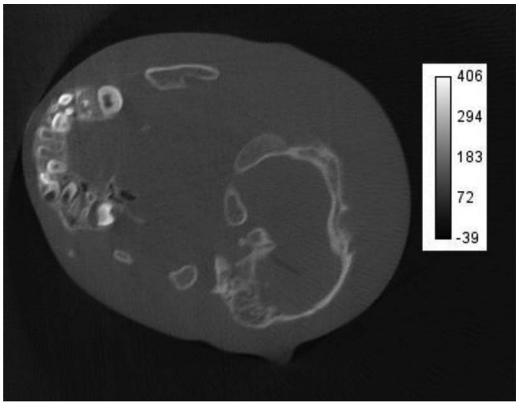

FIG. 4. Slice of the separable footprint reconstructed image.

VII. CONCLUSIONS

In this paper, we presented a novel framework for producing an optimal code structure using an ant colony optimization algorithm. Through our experiments in applying our framework to the RabbitCT platform,[13] we have discovered a better implementation for the 256^3 volume reconstruction, while producing the same results as Ref. 6 for the 512^3 implementation. We have also shown that the optimal implementation can be quite different depending on the hardware. Although it takes some time to find the right optimization parameters, we wish to add that the code produced by the ant colony optimization can be reused for any new CT reconstruction task with the same boundary conditions. Therefore, the optimization overhead is well amortized.

In practice, we have found that producing the super source file takes approximately 30–45 min. We often know what optimizations we want to include. Then it is simply a matter of coming up with a graph structure and annotating the code snippets. Determining what optimizations should be included, however, can be time consuming if the user is not completely familiar with the problem he is trying to optimize or the CUDA programming language. Our future work is therefore directed toward automating the process of generating the super source file. We are also interested in exploring alternative algorithms that could be applied to our framework. If we can find an effective mapping from the non-numerical implementation to a high dimensional space, we can use numerical optimization algorithms like gradient descent to find the optimal implementation over a larger search space.

ACKNOWLEDGMENTS

This work was funded in part by National Science Foundation (NSF) Grant Nos. IIS-1050477, CNS-0959979, and IIS-1117132. The authors also thank Medtronic for partial funding of this work and datasets they used in experiments.

[a]Electronic mail: epapenhausen@cs.sunysb.edu
[b]Electronic mail: zizhen@cs.sunysb.edu
[c]Electronic mail: mueller@cs.sunysb.edu

[1]A. Andersen and A. Kak, "Simultaneous algebraic reconstruction technique (SART): A superior implementation of the ART algorithm," Ultrason. Imaging, **6**, 81–94 (1984).
[2]L. Feldkamp, L. Davis, and J. Kress, "Practical cone-beam algorithm," J. Opt. Soc. Am. **1**(A6), 612–619 (1984).
[3]L. Shepp and Y. Vardi, "Maximum likelihood reconstruction for emission tomography," IEEE Trans. Med. Imaging **1**(2), 113–122 (1982).
[4]Y. Long, J. A. Fessler, and J. M. Balter, "3D forward and backprojection for X-ray CT using separable footprints," IEEE Trans. Med. Imaging **29**(11), 1839–1850 (2010).
[5]W. Xu and K. Mueller, "Learning effective parameter settings for iterative ct reconstruction algorithms," *Proceedings of the International Meeting on*

Fully 3D Image Reconstruction in Radiology and Nuclear Medicine, Beijing, China, 2009.

[6]E. Papenhausen, Z. Zheng, and K. Mueller, "GPU-accelerated back-projecting revisited: Squeezing performance by careful tuning," *Proceedings of the International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine, Potsdam, Germany*, 2011.

[7]H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-based CT reconstruction using the common unified device architecture (CUDA)," in *Proceedings of the IEEE Medical Imaging Conference, Honolulu, HI* (IEEE, Honolulu, Hawaii, 2007), vol. **6**, pp. 4464–4466.

[8]Z. Zheng and K. Mueller, "Cache-aware GPU memory scheduling scheme for CT back-projection," *Proceedings of the IEEE Medical Imaging Conference*, Knoxville, TN, October 2010.

[9]A. Klockner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to GPU run-time code generation," Parallel Comput. **38**(3), 157–174 (2011).

[10]G. Rudy, M. Khan, M. Hall, C. Chen, and J. Chame, "A programming language interface to describe transformations and code generation," in *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing (LCPC'10)* (Houston, TX, October 7–9, 2010), pp. 136–150.

[11]M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," IEEE Trans. Evol. Comput. **1**(1), 53–66 (1997).

[12]M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents," IEEE Trans. Syst., Man, Cybern., Part B: Cybern. **26**(1), 29–41 (1996).

[13]C. Rohkohl, B. Keck, H. G. Hofmann, and J. Hornegger, "RabbitCT—an open platform for benchmarking 3D cone-beam reconstruction algorithms," Med. Phys. **36**, 3940–3944, 2009.

[14]M. Wu and J. Fessler, "GPU acceleration of 3D forward and backward projection using separable footprints for x-ray CT image reconstruction," *Proceedings of the International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine, Potsdam, Germany*, 2011.

[15]L. Westover, "Footprint evaluation for volume rendering," in *Proceedings of the International Meeting on International Conference on Computer Graphics Interactive Techniques* (Dallas, TX, August 6–10, 1990), pp. 367–376.

[16]B. De Man and S. Basu, "Distance-driven projection and backprojection in three dimensions," Phys. Med. Biol. **49**(11), 2463–2475 (2004).

[17]See supplementary material at http://dx.doi.org/10.1118/1.4773045 for Figs. 1 and 2.