Real-Time Simulation of Ultrasound Fields

December 22, 2011

Richard Abrich
richard.abrich@utoronto.ca

Valentin Berbenetz
valentin.berbenetz@utoronto.ca

Matthew Thorpe
matthew.thorpe@utoronto.ca

Abstract

In this paper we develop and analyze three parallelized versions of an ultrasound field simulation algorithm developed by Aguilar et. al. with the goal of achieving a real-time simulation. Parallelization is implemented on the GPU using CUDA, and on the CPU using OpenMP. We compare the performance of each algorithm by varying three different simulation parameters on two sets of hardware with significantly different specifications. We then examine the impact of these variations on the underlying hardware. In the best case, we achieve a 23x speedup over the original CPU implementation.

1 Introduction

1.1 Doppler Ultrasound

Ultrasound is an area of significant interest in the field of medical imaging. It has several advantages over other conventional imaging techniques such as CT and MRI:

- Muscle, bone and soft tissue structures are clearly viewable
- No discomfort or side effects for the patient
- Live imaging provides swift diagnoses
- Equipment is flexible, portable, and widely available

Doppler ultrasound relies on the Doppler Effect, wherein a waveform's frequency varies as a function of the velocity of its source, to determine the velocity of structures of interest within the human body. For example, measuring the velocity of blood cells flowing through an artery allows medical personnel to determine the level of occlusion, or plaque buildup, within the artery.

In order for patients to benefit from this area of study, medical personnel must keep up with continual improvements in ultrasound equipment and techniques. Current training and testing methods, however, are limited by the use of phantom bodies (synthetic organs), which are expensive and highly specific, or the use of high-latency computer simulations, which are unsuitable for real-time training purposes.

The motivation for this project is thus to provide medical personnel with a software tool to gain valuable experience in ultrasound techniques without the need for a patient who is presenting with a condition of interest, or the use of costly phantom bodies. In order for such a tool to be useful, it must exhibit "real-time" performance, or approximately 30 frames per second, as per the NTSC standard for television video playback. [1] This implies a refresh rate of approximately 30ms per frame.

1.2 CUDA Architecture

In this project, computational speedups are obtained by parallelization through Compute Unified Device Architecture (CUDA), a single-instruction multiple-data architecture developed by NVIDIA for general purpose GPU programming.

The CUDA API allows the programmer to group threads into "blocks" of up to three dimensions, which in turn may be grouped into "grids" of up to two dimensions. The sizes of both blocks and grids are limited on current generations of hardware as per Table 1. Differentiation between threads is achieved by referring to the thread's position within a block and/or grid. [2]

Functions written for execution on the GPU are called "kernels". When a multiprocessor on the GPU is presented with one or more thread blocks for execution, it partitions them into "warps", or groups of 32 consecutive threads. Warps execute a single instruction at a time, and thus maximum throughput is achieved when all threads within a warp execute the same instruction. However,

Dimension	Maximum Size	
	Block	Grid
X	512	65536
Y	512	65536
Z	64	1
Total	512	65536^2

Table 1: CUDA block and grid dimension restrictions. Note that a block's dimensions may not each be set to the maximum allowable value simultaneously, as this would violate the constraint on the total size.

since each thread within a warp has its own instruction pointer, this allows them to diverge, forcing other threads to wait until a synchronization event occurs. In general, care should be taken to ensure that threads within a warp seldom diverge so as to increase throughput.

The CUDA memory model consists of three levels. Each thread has its own private memory, each block has shared memory visible to all threads within the block, and all threads in all blocks have access to the same global memory. Global memory and local memory are both off-chip, and are therefore the slowest (approximately 100x slower than shared memory).

Each multiprocessor (core) within a CUDA device also has a limited number of registers for local variables which, once full, forces the multiprocessor to put local variables into "local" memory. Therefore, shared memory should be utilized wherever possible. Furthermore, care should be taken such that consecutive threads access consecutive locations in memory. This allows the multiprocessor to "coalesce" memory accesses into a single operation. This applies to both shared and global memory.

When a warp stalls on a memory access, the multiprocessor begins execution on another warp, thereby hiding memory access times. The ratio of the number of active warps per multiprocessor to the maximum number of possible active warps is called the occupancy. The higher the occupancy, the more likely it is for the multiprocessor to have useful work to do, and the better the performance. [3]

2 Related Work

Field II is a popular free software tool for simulating ultrasound fields using convolutional methods. While this affords it a high degree of accuracy, its latencies make it unsuitable for performing real-time simulations. Furthermore, inherent data dependencies make it an unsuitable candidate for parallelization.

Shams et. al. developed a parallel algorithm for computing the spatial impulse response (SIR) of an ultrasound source using CUDA, which achieved significant speedup over the Field II implementation of this calculation step [4]. But although it is required for simulating an ultrasound field, the SIR calculation does not represent the bulk of computation in an ultrasound simulation application.

Aguilar et. al. developed an alternate method for simulating ultrasound fields that relies on a summation instead of a convolution [5]. This method models the ultrasound source as an array of acoustic monopoles, each of which emits an identical sinusoid, or base pulse. Points in space at which we are interested in computing the acoustic field are known as acoustic scatterers. The field which results from a single monopole at a single scatterer is calculated as a function of the Euclidean distance between that monopole and scatterer. The total field at each scatterer is simply the summation of the individual contributions of each monopole.

While this method reduces the latencies associated with simulation by several orders of magnitude, it can still take over a minute to compute a single frame on a high-end desktop. However, the main advantage of this method is that it also removes several (but not all) data dependencies, thus lending itself very well to parallelization.

The most computationally expensive portion of Aguilar's method is contained within a triply nested for loop which iterates through all of the monopole/scatterer pairs, and accumulates the contribution of the entire pulse length at each scatterer. This calculation exhibits a Write-After-Write dependency between monopoles where their contributions to a single scatterer overlap, as indicated in Figure 1.

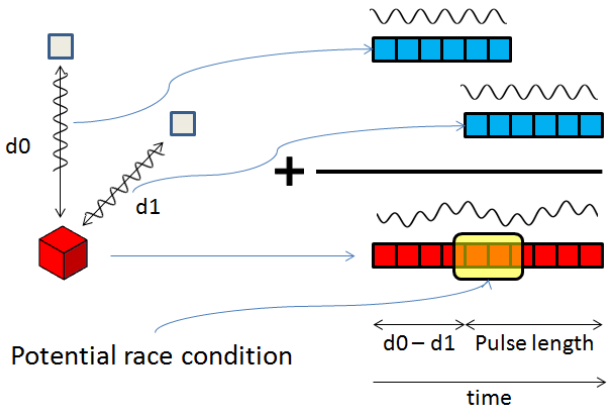

Figure 1: A race condition occurs where the contributions of different monopoles overlap in a single scatterer

The rest of this paper focuses on the parallelization of this portion of Aguilar's method, using the software

framework from a previous attempt [6] as a basis.

3 Proposed Solution

Variables in CUDA can be mapped to one of five dimensions: block height, width, and depth; and grid height and width. In our case, we have three variables: the number of scatterers, the number of monopoles, and the length of the pulse. Thus, the number of possible combinations for variable assignments is:

$$5P3 = \frac{5!}{(5-3)!} = 5 * 4 * 3 = 60$$

However, since we are limited by the number of threads per block, and the number of blocks per grid, this figure represents a lower bound, as variables may also need to be subdivided and executed on separately. It should be noted, however, that not all of these combinations are necessarily feasible due to the WAW dependency discussed in Section 2.

For the rest of this paper, we will limit our discussion to only two GPU configurations, as these were the most intuitive and practical out of those that we considered. We shall also consider a parallelized algorithm on the CPU using OpenMP.

3.1 CPU: OpenMP

A coarsely parallelized version of the algorithm was implemented on the CPU using OpenMP. This involved parallelizing the outermost loop, which iterates over monopoles.

The monopoles were split amongst separate loops such that those writing to adjacent locations in the accumulator were operated on within the same thread. This behaviour mitigated the WAW dependencies at the accumulator by serializing writes to each memory location.

3.2 GPU: Naive

In keeping with the limitations outlined in Table 1, the most obvious choice for block and grid dimension assignments are shown in Table 2.

	Grid	Block
X	# of monopoles	# of pulse elements
Y	# of scatterers	1
Z	1	1

Table 2: Dimension assignments for Naive GPU implementation

While this configuration does not violate the aforementioned dimentionality restrictions, it fails to take into account the WAW dependencies on the accumulator discussed earlier.

One possible workaround would be to allocate separate accumulators for each monopole, and then perform a reduction summation on the accumulators. However, the number of monopoles makes the space required for these accumulators infeasible on modern hardware. Therefore, two modifications on this idea are outlined in the next two sections.

3.3 GPU: Loop

The block and grid dimensions for this implementation are shown in Table 3. In order to avoid the write dependence between monopoles, this algorithm calculates the contribution of each monopole to each scatterer one monopole at a time. Looping through each monopole is performed within the kernel, so as to reduce kernel launch overhead. This scheme eliminates the need for multiple accumulators.

	Grid	Block
X	1	# of pulse elements
Y	# of scatterers	1
Z	1	1

Table 3: Dimension assignments for GPU Loop implementation

All threads in a single block correspond to elements in a single pulse. As such, several data arrays required for the acoustic amplitude calculations are shared amongst threads in a block. Thus, at the beginning of kernel execution, a single thread in each block loads the data to be used by other threads in the block into shared memory, after which point threads are no longer required to read from global memory until the final write of the amplitudes into the accumulator. This reduces the impact of the latencies associated with global memory accesses on the algorithm's throughput.

3.4 GPU: Reduction

The block and grid dimensions for this implementation are shown in Table 4. This scheme avoids the WAW dependency on the accumulator by following a scheme similar to that proposed for the naive GPU implementation. However, since there is not enough memory to allocate accumulators for all monopoles at once, this implementation queries the device at run-time to determine the amount of available memory, and then determines the resulting maximum number of monopoles for which copies

of the necessary data structures can be allocated. This includes data structures from which only reads occur, so as to prevent serialization of reads, as described in section 1.2.

	Grid	Block
X	Max # of monopoles per iteration	# of pulse elements
Y	# of scatterers	1
Z	1	1

Table 4: Dimension assignments for GPU Reduction implementation

The algorithm then loops through as many batches of monopoles as necessary, with concurrent monopoles writing to their own unique accumulator array. Once completed, the separate accumulators are reduced into a single accumulator through a reduction summation.

As in the GPU Loop implementation, looping is performed within the kernel so as to reduce kernel launch overhead. This implementation also utilizes shared memory in a manner similar to the GPU Loop implementation so as to reduce the impact of global memory latency on throughput.

This scheme exhibits significant overhead over the GPU Loop implementation due to a) the allocation of many duplicate memory structures, and b) the reduction summation.

4 Methodology

The tests outlined in this section consist of varying the sizes of the three parameters at our disposal: the number of monopoles, the number of scatterers, and the pulse length. The pulse length was increased by varying the number of cycles in each pulse from 1 – 20, the number of scatterers was varied from 10 – 10,000, and the number of monopoles was varied from 9 – 1681. Note that as the the number of each parameter increases, so does the accuracy of the simulation.

Table 5 lists the values for the three parameters in our experiments. Varying each parameter individually allows us to understand how adjusting the accuracy of the system affects the overall performance.

Testing was performed on two different sets of hardware, low-end and high-end, as outlined in Tables 6 and 7, respectively. This variation was performed in order to ensure that our proposed solution did not favour a specific hardware configuration, but rather provided real speedups for any hardware configuration. All implementations (including the original CPU version) were compiled using O2 compiler optimizations.

Table 6: Low-end system configuration

	CPU	GPU
Name	Atom D525	Nvidia Ion 2
Speed	1.8GHz*	1.09GHz
Cores	2 (4)**	16

* Hyper-threading enabled
** Number of virtual cores due to hyper-threading support

Table 7: High-end system configuration

	CPU	GPU
Name	Core i7-2630QM	GeForce GTX470
Speed	2.0GHz*	1.215GHz
Cores	4 (8)**	448

* Hyper-threading enabled
** Number of virtual cores due to hyper-threading support

5 Results

In all three of the parameter variation experiments below, GPU Reduction exhibits poor performance for small workloads. This is due to a) the overhead associated with allocating duplicate datastructures, b) the overhead associated with performing the reduction summation (which requires an additional kernel invocation), and c) an increased number of local variables per thread, which exceeds the number of registers per multiprocessor. (This forces the compiler to allocate variables in local memory, which as stated in section 1.2, is approximately 100x slower than shared memory.)

Furthermore, the overhead involved in copying memory between the CPU and GPU before and after a kernel launch causes poor performance for both GPU Loop and GPU Reduction for small workloads.

Additionally, the CPU multithreaded implementations are able to achieve super-linear speedups due to caching. More specifically, threads operating on the same pulse are able to utilize data within the cache, thereby reducing the overall number of fetches from memory.

5.1 Varying Pulse Length

The first series of experiments involved varying the pulse length, which was done by increasing the number of cycles within each pulse. On the CPU, this results in more work per thread, while on the GPU, this results in a greater amount of threads per block. Performance degradation was relatively constant for each increase on both low-end (Figure 2) and high-end (Figure 3) hardware, with similar trends exhibited on each.

Experiment	Parameter	Pulse Length (# of Cycles)	Number of Scatterers	Number of Monopoles
1	Pulse Length	1	1000	441
2		5		
3		10		
4		20		
5	# of Scatterers	10	10	441
6			100	
7			1000	
8			10000	
9	# of Monopoles	10	1000	9
10				121
11				441
12				1681

Table 5: Experimental parameters

At 5 cycles per pulse, GPU Loop yields better performance than OpenMP because of the greater number of threads available and the light weight nature of threads in GPU architecture. Similarly, GPU Reduction begins yielding comparable performance to OpenMP when the reduction overhead corresponds to a smaller fraction of the total execution time at 10 cycles per pulse.

Due to the occupancy ratio discussed in section 1.2, a threshold in the number of threads capable of doing useful work in GPU Loop is reached at around 10 cycles per pulse, causing a smaller increase in performance for increasing pulse length thereafter. Although GPU Reduction reaches this same threshold, it processes multiple monopoles in parallel, allowing it to continue processing subsequent monopoles in the same group, even if calculation for one monopole stalls.

For a 20-cycle pulse, GPU Loop achieved a 1.5x speedup over OpenMP due to its highly parallelized nature, while GPU Reduction saw a 1.2x slowdown due largely to the substantial overhead described earlier. Nevertheless, this is still an improvement over the 3.9x slowdown for a single-cycle pulse, and over the 1.3x slowdown for the 10-cycle pulse. This indicates that for scenarios involving substantially larger pulses on low-end hardware, both GPU implementations provide speedups over CPU multi-threading with OpenMP.

All three multithreaded implementations provide at least 4.0x speedup for the largest pulse length over their single-threaded counterpart on low-end hardware.

Similar trends are exhibited on the high-end hardware, where the overhead causes similar performance for OpenMP and GPU Loop, and poor performance for GPU Reduction at short pulse lengths, while GPU Reduction begins to overcome this overhead at around a 10-cycle pulse. For a 20-cycle pulse, all implementations had at least a 3.6x speedup over the naive single-threaded im-

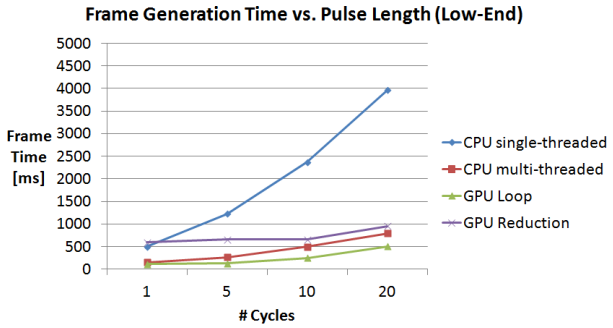

Figure 2: Varying Pulse Length on Low-End Hardware

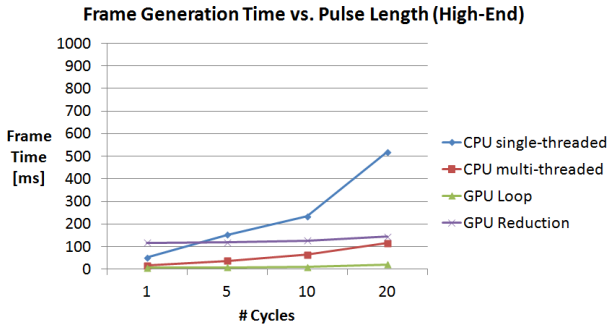

Figure 3: Varying Pulse Length on High-End Hardware

plementation. Due to faster relative cores available on the high-end hardware, GPU Loop achieved a speedup of 5.8x over OpenMP, while GPU Reduction saw a 1.3x slowdown on the 20-cycle pulse.

As on the low-end hardware, GPU Reduction on the high-end hardware saw a greater slowdown (1.9x) when testing with a 10-cycle pulse, supporting the claim that this slowdown is largely due to the reduction and allocation overhead.

It should be noted that the shortest frame generation time achieved on low-end hardware was 108ms for a single-cycle pulse, several times the minimum time required to be considered real-time. Unlike on the low-end hardware, however, GPU Loop on the high-end hardware is able to generate frames in 20ms for the single-cycle pulse, which is under the threshold for being considered real-time.

5.2 Varying Number of Scatterers

The second series of experiments involved varying the number of scatterers. On the CPU, this results in more work per thread, while on the GPU, this results in larger Grid sizes, which in turn corresponds to a greater number of warps to be executed by each multiprocessor.

For 10,000 scatterers, our low-end hardware (Figure 4) performed very poorly, with all implementations taking at least 2.5 seconds to generate a single frame. Similar results are seen on the high-end hardware (Figure 5). As expected, the initialization overhead associated with the GPU implementations dwarfs potential performance improvements for both GPU Loop and GPU Reduction in the 10-scatterer test case. But for a large number of scatters, GPU Reduction overcomes this overhead, reducing from a slowdown of 7.1x in the 10-scatterer case to only 1.4x in the 10,000 scatterer case.

Similarly to increasing the pulse length, increasing the number of scatterers causes the same performance trend between GPU Loop and GPU Reduction. As explained in the previous section, GPU Loop is more easily affected by occupancy, leading to slight performance degradation at a large number of scatterers. At 1,000 scatterers, GPU Loop has a speedup over GPU Reduction of 13.0x, while at 10,000 scatterers the speedup decreases to 4.1x.

Both GPU implementations take advantage of lightweight threads on the GPU and use fine grained parallelization, lessening the burden of an increasing numbers of scatterers. However, OpenMP cannot take advantage of lightweight threads and thus each thread has an increasing amount of computation. For GPU Loop, this advantage is lost due to occupancy, however GPU Reduction sees continued performance improvement. At 1,000 scatterers GPU Loop has a speedup of 6.4x and GPU Reduction has a slowdown of 2.0x compared to OpenMP, while at 10,000 scatterers, GPU Loop and GPU Reduction have speedups of 4.8x and 1.2x respectively.

Although GPU Loop is still able to generate frames in real time at 1,000 scatterers (with 10ms per frame), increasing the accuracy by scaling number of scatterers up to 10,000 causes all three implementations to be above the threshold for real-time simulation.

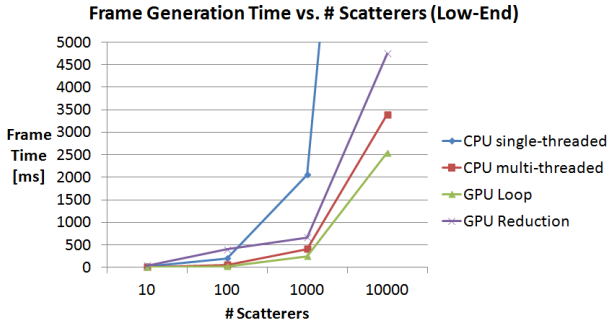

Figure 4: Varying # Scatterers on Low-End Hardware

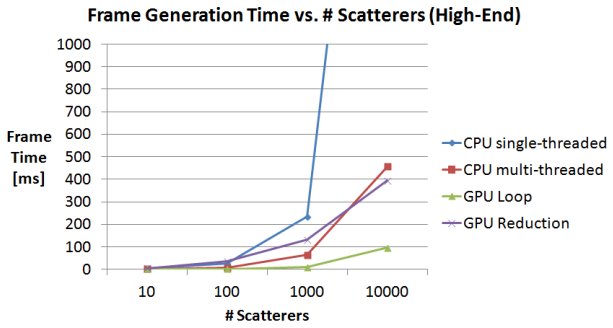

Figure 5: Varying # Scatterers on High-End Hardware

5.3 Varying Number of Monopoles

The third series of experiments involved varying the number of monopoles. As the number of monopoles increases, the GPU Reduction algorithm tends to generate a better speed up when compared to both the multithreaded CPU and the GPU Loop. As can be seen in Figures 6, at 441 monopoles, GPU Reduction has a slowdown of 2.65x when compared with the Loop, and 1.64x when compared with the CPU multithreaded version. However, as the number of monopoles is increased to 1681, the slowdown with respect to the Loop implementation is reduced to 1.4x, and now has a 1.14x speedup when compared with the CPU multithreaded version.

The underlying reason for this has to do with the occupancy being higher in the reduction algorithm when

compared with the loop algorithm. In the event of a warp stalling on a memory access, the multiprocessor can continue doing work on warps associated with other monopoles until that warp is completed, or until it also experiences a stall.

The coarse-grain multithreaded CPU implementation begins to degrade in performance more quickly as the number of monopoles increases relative to the GPU implementations, because it is limited in the number of threads. This means that as the number of monopoles increases, each CPU thread needs to take on much more work than a GPU thread.

When varying the number of monopoles on the high-end system (Figure 7), we notice that the Reduction algorithm only increases by 11ms, from 123ms to 134ms, when we move from 441 to 1681 monopoles. As mentioned previously, the bulk of this computation is due to the overhead associated with allocation and reduction of multiple accumulators. This shows that the actual computation time is only a small fraction of the overall frame computation time. On the other hand, the Loop algorithm increased from 10ms to 37ms. This is because this implementation has far less overhead than the Reduction.

Figures 6 also shows that the CPU multithreaded implementation begins to degrade in performance more severely on the low-end system as it takes on more work. This difference in performance between the low-end and high-end systems is attributable to the high-end system having twice as many cores, thereby decreasing the potential for stalled warps.

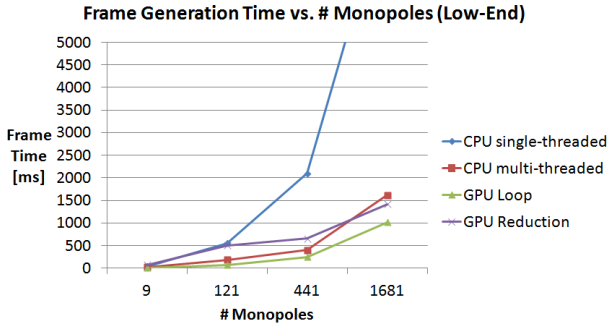

Figure 6: Varying # Monopoles on Low-End Hardware

6 Conclusion

In this paper, we showed that significant speed-ups to Aguilar's algorithm are achievable through parallelization on the GPU to the point of approaching real-time frame generation. On the baseline test case of 10 cycles per pulse, 1000 scatterers, and 441 monopoles, our GPU Loop implementation achieves a speedup of 23x over the

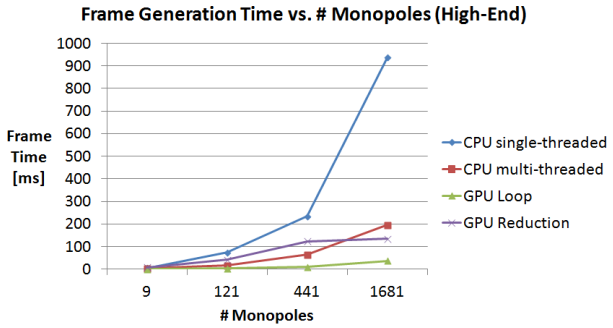

Figure 7: Varying # Monopoles on High-End Hardware

original CPU implementation, resulting in a frame generation time of 10ms. However, increasing the number of scatterers or monopoles so as to achieve more accurate results causes frames to no longer be generated in real time. Furthermore, these times do not include the time required to render the frame in OpenGL.

Nevertheless, these algorithms are designed to scale according to hardware capacity, and as such, future hardware is expected to demonstrate even better performance, with little to no modification of the source code required.

Possible improvements to these algorithms include low-level tweaking such as warp synchronization and optimization of PTX machine code, as well as higher-level concurrency such as multiple asynchronous kernels and multiple GPUs. Future work should also include research into the impact of frame rendering on performance.

References

1 Apple Corporation, Working with NTSC Video, http://documentation.apple.com/en/cinematools/usermanual/index.html#chapter=2%26section=5, accessed 16 October 2011.

2 CUDA C Programming Guide version 4.0, http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf, accessed 13 December 2011

3 CUDA C Best Practices Guide version 4.0, http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf, accessed 14 December 2011.

4 An algorithm for efficient computation of spatial impulse response on the GPU with application in ultrasound simulation, Biomedical Imaging: From Nano to Macro, 2011 IEEE International Symposium, March 30 2011–April 2 2011, p. 45 – 51

5 Luis A. Aguilar, David A. Steinman, Richard S.C. Cobbold, On the Synthesis of Sample Volumes for Real-Time Spectral Doppler Ultrasound Simulation, 2010.

6 Richard Abrich, Vajira Sarathchandra, Real-time Simulation of Ultrasound Fields, 2011.