Ultrasound Simulation on the Cell Broadband Engine using the Westervelt Equation

Andrew A. Haigh[1], Bradley E. Treeby[2] and Eric C. McCreath[1]

[1] Research School of Computer Science, The Australian National University
[2] Research School of Engineering, The Australian National University
ahaigh1@gmail.com, bradley.treeby@anu.edu.au,
eric.mccreath@anu.edu.au

Abstract. The simulation of realistic medical ultrasound imaging is a computationally intensive task. Although this task may be divided and parallelized, temporal and spatial dependencies make memory bandwidth a bottleneck on performance. In this paper, we report on our implementation of an ultrasound simulator on the Cell Broadband Engine using the Westervelt equation. Our approach divides the simulation region into blocks, and then moves a block along with its surrounding blocks through a number of time steps without storing intermediate pressures to memory. Although this increases the amount of floating point computation, it reduces the bandwidth to memory over the entire simulation which improves overall performance. We also analyse how performance may be improved by restricting the simulation to regions that are affected by the transducer output pulse and that influence the final scattered signal received by the transducer.

Keywords: ultrasound simulation, Westervelt equation, Cell Broadband Engine, parallelization.

1 Introduction

The simulation of realistic ultrasound signals is important in a number of fields, including ultrasound system design and development [1], the delivery of therapeutic ultrasound [2], and the registration of diagnostic ultrasound images with other imaging modalities [3]. However, ultrasound simulation is a computationally intensive task due to the large number of grid points and time steps required to accurately replicate typical biomedical scenarios. For example, the central frequency of a diagnostic ultrasound transducer can range from $2 - 15$ MHz, with depth penetrations from cms to tens of cms. To discretize domains of this size, a 2 dimensional finite-difference time-domain (FDTD) simulation can require grid sizes in excess of 1000×1000 grid points [4, 5]. Similarly, the simulation of a single scan line in which waves propagate from the transducer into the medium and back can require more than 6000 time steps [6]. This is increased further if the simulation of nonlinear harmonics is required, or if the simulations are performed in 3 dimensions. This is to the point where merely storing the pressure values at each grid point becomes difficult on modern desktop computing systems. The computational task is challenging both from the perspective of the number of floating point operations, and also the transfer of data to and from the CPU.

Recently, nonlinear ultrasound simulation on general-purpose graphics processing units (GPGPUs) have been attracting attention [5–7]. This has helped with the floating point operations, however, memory bandwidth is still a bottleneck. Our research extends the GPGPU implementation discussed by Karamalis et al. [6], and explores how the Westervelt Equation could be used to simulate ultrasound on the Cell processor. In this case, the Westervelt equation is solved with a FDTD scheme which can take advantage of important aspects of the Cell's hardware, including multiple cores, SIMD calculations, and asynchronous memory retrieval/storage. We have designed and implemented a simulator on the Cell hardware, and explored how the memory transfers may be reduced by dividing the region up into blocks of pressure samples and stepping these multiple steps in time. This approach is shown to improve the performance and we believe is novel within the ultrasound simulation field. We have also explored how the number of grid points used in the simulation can be limited to those that are affected by the ultrasound signal transmitted by the transducer, or that influence the scattered ultrasound signal that is subsequently received.

We have evaluated our approaches on the Cell microprocessor as it has been shown that the Cell can be used effectively for computation-intensive applications [8–10]. In particular the Cell has been used effectively for FDTD calculations [11, 12]. However, the approaches presented in this paper are not limited to the Cell and may be applied to other architectures such as GPGPUs or more standard multi-core/cluster architectures. We envisage that similar improvements in performance would also be gained on these systems.

This paper is organised as follows. In Section 2 we give the Westervelt equation along with the finite difference discretization used in our implementation. In Section 3 we overview the salient features of the Cell processor. Section 4 gives the approach taken in implementing our simulation. In Section 5 our results are reported. These focus on understanding the improvements gained by applying the multiple time steps and only simulating required regions. Finally in Section 6 we provide a conclusion along with a discussion of possible future research.

2 Westervelt Equation

The lossy Westervelt Equation for a thermoviscous fluid is given by [13]

$$\nabla^2 p - \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} + \frac{\delta}{c^4}\frac{\partial^3 p}{\partial t^3} + \frac{\beta}{\rho_0 c^4}\frac{\partial^2 p^2}{\partial t^2} = 0, \tag{1}$$

where p is the acoustic pressure (Pa), c is the propagation speed ($\mathrm{m\,s^{-1}}$), ρ_0 is the ambient density ($\mathrm{kg\,m^{-3}}$), δ is the diffusivity of sound ($\mathrm{m^2\,s^{-1}}$), and β is the coefficient of nonlinearity. The first two terms are equivalent to the conventional linearized wave equation, the third term accounts for thermoviscous absorption, and the fourth term accounts for cumulative nonlinear effects.

Here, this equation is solved numerically using the FDTD method on a rectangular grid. The grid has spacing Δx and Δy in the x and y dimensions (m), where $\Delta x = \Delta y$. The time steps are equally spaced and given by Δt (s). $p^n_{i,j}$ denotes the variation from the background acoustic pressure at time step n at grid point (i, j). The gradients

are approximated using finite differences that are fourth-order accurate in space and second-order accurate in time [14]. This is the same as the schemes used in [6, 15]. The finite differences generated by this algorithm and used in the code are:

$$\frac{\partial^2 p}{\partial t^2} \approx \frac{1}{\Delta t^2}\left(p_{i,j}^{n+1} - 2p_{i,j}^{n} + p_{i,j}^{n-1}\right), \tag{2}$$

$$\frac{\partial^3 p}{\partial t^3} \approx \frac{1}{2\Delta t^3}\left(3p_{i,j}^{n+1} - 10p_{i,j}^{n} + 12p_{i,j}^{n-1} - 6p_{i,j}^{n-2} + p_{i,j}^{n-3}\right), \tag{3}$$

$$\frac{\partial^2 p}{\partial x^2} \approx \frac{1}{12\Delta x^2}\left(-p_{i+2,j}^{n} + 16p_{i+1,j}^{n} - 30p_{i,j}^{n} + 16p_{i-1,j}^{n} - p_{i-2,j}^{n}\right), \tag{4}$$

$$\frac{\partial^2 p}{\partial y^2} \approx \frac{1}{12\Delta y^2}\left(-p_{i,j+2}^{n} + 16p_{i,j+1}^{n} - 30p_{i,j}^{n} + 16p_{i,j-1}^{n} - p_{i,j-2}^{n}\right). \tag{5}$$

The $\frac{\partial^2 p^2}{\partial t^2}$ term is calculated by making use of the chain rule and product rule:

$$\frac{\partial^2 p^2}{\partial t^2} = 2\left(\left(\frac{\partial p}{\partial t}\right)^2 + p\frac{\partial^2 p}{\partial t^2}\right), \tag{6}$$

where the temporal gradients are respectively computed using explicit third-order and second-order accurate backward finite differences

$$\frac{\partial p}{\partial t} \approx \frac{1}{\Delta t}\left(\frac{11}{6}p_{i,j}^{n} - 3p_{i,j}^{n-1} + \frac{3}{2}p_{i,j}^{n-2} - \frac{1}{3}p_{i,j}^{n-3}\right), \tag{7}$$

$$\frac{\partial^2 p}{\partial t^2} \approx \frac{1}{\Delta t^2}\left(2p_{i,j}^{n} - 5p_{i,j}^{n-1} + 4p_{i,j}^{n-2} - p_{i,j}^{n-3}\right), \tag{8}$$

(the use of a backward difference avoids $p_{i,j}^{n+1}$ (etc) terms and makes solving the final equation easier). Combining these equations with (1) using a second-order accurate finite difference scheme for the remaining temporal derivative then gives

$$\begin{aligned} p_{i,j}^{n+1} = \left(c^{-2}\Delta t^{-2} - \frac{3}{2}\delta c^{-4}\Delta t^{-3}\right)^{-1} \Bigg(&\nabla^2 p + p_{i,j}^{n}(2c^{-2}\Delta t^{-2} - 5\delta c^{-4}\Delta t^{-3}) \\ &+ p_{i,j}^{n-1}(6\delta c^{-4}\Delta t^{-3} - c^{-2}\Delta t^{-2}) - p_{i,j}^{n-2}(3\delta c^{-4}\Delta t^{-3}) \\ &+ p_{i,j}^{n-3}(\frac{1}{2}\delta c^{-4}\Delta t^{-3}) + \frac{\beta}{\rho_0 c^4}\frac{\partial^2 p^2}{\partial t^2}\Bigg), \end{aligned} \tag{9}$$

where in 2 dimensions $\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}$. Note that c is actually dependent on (i, j) as it varies depending on the properties of the medium at each position. For simplicity we assume reflecting boundary conditions (i.e. $p_{i,j}^{n} = 0$ for all (i, j) which lie outside the grid.)

3 The Cell and the PS3

The Cell processor and the PS3 have been extensively described in articles such as [16]. A brief overview of the salient aspects is now given. The PS3 has a 3.2GHz Cell

processor along with 256MB of main memory which is Rambus XDR DRAM. The Cell Broadband Engine (CBE) is a microprocessor developed by Sony, Toshiba and IBM [16] which contains 9 cores. These cores are composed of a single Power processor element (PPE) intended to be used for coordinating the actions of the other cores, and eight Syngergistic processor elements (SPEs) which are RISC SIMD processor elements and are optimised for numerical computation [17, p. 34].

The PPE is a two-way multi-threaded core with 512kB of L2 cache. Linux may be installed and run on the PS3, although a hypervisor restricts access to the RSX 'Reality Synthesizer' graphics processing unit and also disables 2 of the 8 SPEs [18, p. 7]. Ubuntu gutsy (7.10) running Linux 2.6.20 64 bit SMP version has been installed on our test machine. Figure 1 depicts a block diagram of the effective architecture when the system is run with the hypervisor.

Each SPE has 128 128-bit registers which can be used to perform SIMD calculations. The SIMD instruction set [19, Section 2] contains a large number of useful instructions. These include operations that may be performed on vectors of 4 floats (contained in a single 128-bit register). An example of such an operation is to construct a new vector using 2 floats from one vector and 2 floats from another vector. These operations are pipelined and may be issued on every clock cycle. The arithmetic/logic operations are done in the even side of the pipeline. The odd side of the pipeline includes operations such as load/store between registers and the local store (LS). A multiply-add operation enables a maximum of 8 floating point operations per cycle thus giving a maximum of 25.6GFlops per SPE, although for most operations this maximum is 12.8GFlops.

The Cell has an internal bus, called the Element Interconnect Bus (EIB), which enables the SPEs to communicate with each other, the PPE, and the memory interface. The effective bandwidth of the EIB is 96 bytes per cycle [16, Figure 1].

Each SPE contains 256kB of LS to be used for data, instructions and stack and this is the only memory that can be accessed directly by each SPE; of this storage, 240kB is available for data. Data is transferred between main memory and local store via Direct Memory Access (DMA) requests [16, p. 595]. These accesses can be initiated by either the SPE or PPE and are asynchronous and can be barriered/fenced against each other [19, p. 57]. Each request can transfer at most 16kB.

Each SPE channel to the EIB is limited to 25.6GB/s, moreover, the transfers between the EIB and main memory is also limited to 25.6GB/s, this becomes a bottleneck when only a few operations are needed to be performed on each float that is transferred to/from main memory. A particular type of request that we make heavy use of is a 'scatter and gather' request where multiple discontiguous area of main memory are loaded into a contiguous area of local storage. Correct alignment of data is important for the performance of the system.

4 Approach

Solving the FDTD scheme is performed solely by the (available) SPEs, with the PPE responsible for delegating tasks to the SPEs and coordinating their actions. Initially the PPE allocates $1/6$th of the total spatial region (consisting of a large number of smaller

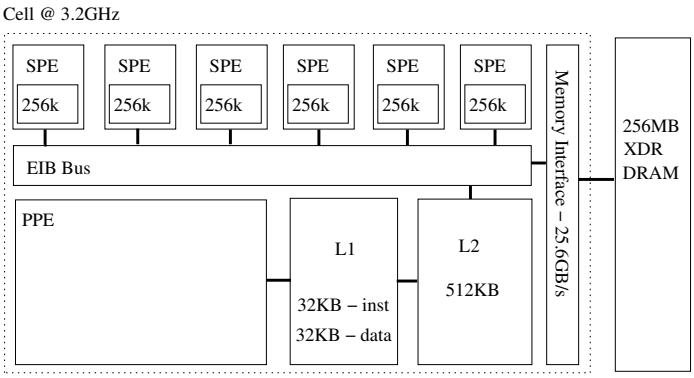

Fig. 1. Block diagram of the PS3 architecture

square regions we call 'blocks') to each SPE during the process of thread creation. Mailboxes are used for synchronisation; each SPE places a message in its outbound mailbox when it has completed an entire timestep in its allocated region and then waits for a message in its inbound mailbox before restarting the calculations for the same region (moved on by a number of timesteps). Although this minimizes the amount of synchronisation and simplifies the implementation, a downside of this approach is that the load on SPEs may become unbalanced due to contention of main memory accesses, resulting in some SPEs waiting while other still have calculations to perform.

The SPE performs the calculation for a single block at a time. The pressures at all grid points for the previous 4 timesteps in that block and all 8 neighbouring blocks are loaded. The pressures are then stepped forwards by as many timesteps as possible. The final pressures and the pressure at the transducers for all intermediate timesteps (i.e. the received signal) are stored in main memory.

Our implementation uses a number of standard techniques [9, 12, 10] to help improve performance, these include SIMD, double buffering, and loop unrolling. In addition to these standard approaches, we also explore how multiple timesteps (Section 4.4) and ignoring regions (Section 4.5) may be used to improve performance. The various techniques used in the implementation are now given.

4.1 SIMD

It is relatively straightforward to use the Cell's SIMD capabilities to improve the performance of the calculations. Since our calculations are performed in single precision, four floats can fit in a single register and so the FDTD scheme can be solved at four grid points simultaneously. The SIMD instruction set contains fused multiply and add instructions which are used in the main computation loop.

There is some extra overhead (compared to the scalar version) due to permute-style operations which are required to approximate the $\frac{\partial^2 p}{\partial y^2}$ term. The permute operations are required because misaligned (not on 128 bit boundary) loads can not be performed directly.

4.2 Double buffering

Double buffering was used to simultaneously calculate the solution of the FDTD scheme and perform the DMA accesses to retrieve the next block (and its neighbours) where calculation should be performed. The cost of the DMA latency is reduced in this way.

However, the downside is that the amount of data that has to be stored simultaneously in the local storage is doubled, which reduces the size of blocks that can be used, and consequently, the number of timesteps that can be performed in between memory accesses.

4.3 Manual loop unrolling

A technique that was found to be effective in improving performance was to manually unroll the main computation loop (that performs the solving of the FDTD scheme) by a factor of 3. This allows extra instructions to be interleaved during compiler optimization and reduces stalls due to dependencies.

4.4 Multiple timesteps in between memory access

In order to minimize the amount of DMA required, the scheme is solved for multiple time steps in a single block before writing the result back to main memory. An area of 3×3 blocks ($3d \times 3d$ grid points) is loaded into main memory, with the intention of calculating the pressures for the central block ($d \times d$ grid points) over a number of timesteps. Under a fourth-order spatially accurate scheme, no more than $\lfloor \frac{d}{2} \rfloor$ timesteps can be calculated at a time, since the region that can be stepped forward in time shrinks 2 grid points with each time step (since pressures cannot be calculated on the boundary of the region).

Other CBE programmers (including [9]) have found it necessary to ensure that large amounts of work is performed on data that is loaded from main memory to avoid being limited by the memory bandwidth. Work on performing linear algebra computations on the CBE [8] uses similar techniques.

Using this timestep approach, it is advantageous to do calculations on blocks that are as big as possible. This maximizes the number of timesteps that can be performed in between memory accesses. Here, blocks of size 24 x 24 were used, allowing 12 time steps to be performed in between memory accesses. Taking into account double buffering, the amount of memory used in our implementation to store the pressures and an index to wavespeeds at an SPE is the equivalent of 51840 floats, or about 210 kB (out of the total 240 kB local storage that is available).

Performing multiple timesteps comes at the cost of extra computation (since some pressures are calculated more than once). Let t denote the number of time steps performed in between memory accesses. Then $d \geq 2t$, and ideally $t = \frac{d}{2}$. The total number of times we apply the FDTD formula in between memory accesses is:

$$W = \sum_{i=0}^{t-1} (d + 2 \times 2i)^2 = d^2 t + 4dt(t-1) + \frac{8t}{3}(t-1)(2t-1) \qquad (10)$$

compared to td^2 for the same amount of useful work if only one timestep is performed at a time. It can be shown that the relative amount of extra work that is done is bounded by

$$\frac{\text{extra work}}{\text{total work}} = \frac{W - d^2t}{d^2t} \leq \frac{10}{3} \tag{11}$$

i.e., for large t, the extra work done is about three times the amount of work performed if a single timestep is performed in between memory accesses.

Let T be the number of timesteps backward required to solve the FDTD scheme (this depends on the FDTD scheme used; $T = 4$ in our case). In this implementation the calculation requires that $9d^2(T+1)$ floats are loaded to calculate t timesteps forward in some block. If one timestep is performed at a time, $(T+1)(d+4)^2$ floats are loaded to calculate one timestep forward in some block. Therefore, the amount of loads performed is reduced for $t \geq 9$ by a factor of approximately $\frac{9}{t}$. In our case, for $t = 12$, the total amount of loads is reduced by about $1/4$.

After each block of t timesteps is completed, the previous $T \leq t$ timesteps worth of pressures at each point must be stored in main memory. When a single timestep is performed at a time, 1 timestep worth of pressures at each point must be stored (since the previous ones have not changed). This means that the total amount of stores that are performed is reduced by a factor of $\frac{t}{T}$ under the new scheme (a factor of 3 in our case). In our implementation stores are more costly than loads because the individual stores operate on quite small pieces of data.

4.5 Ignore empty regions

Regions in which the acoustic pressure is approximately zero, or where the local waveforms cannot affect the final signal received by the transducer do not have to be considered in the calculations. At the start and end of the computation, only regions very near the transducer need to be simulated. Since the number of time steps is chosen to be roughly enough for a pulse to reflect off the furthest wall and return, only a small fraction of the timesteps require the entire region to be simulated. Here, this is implemented conservatively by increasing the height of the simulated region by one grid point per timestep, until the entire grid is being simulated, and reversing this process toward the end of the simulation. The effective width of the grid remains constant because in our implementation the array of transducers is aligned in this direction.

5 Results

The proposed method was implemented in C. The code that runs on the PPE (and organises the calculation) is about 400 lines of code in total. The SPE code which performs the actual calculation (including memory accesses and double buffering) is about 350 lines.

For the simulation we used $\Delta x = \Delta y = 5.0 \times 10^{-5}$m and $\Delta t = 3.0 \times 10^{-9}$s giving a CFL number[3] of $\sim$0.1, this provides a good balance between accuracy and performance. A 3 cycle tone burst with a centre frequency of 3 MHz was used. This provides 10 grid points per wavelength at the centre frequency. The medium parameters were set to $\beta = 6$, $\rho_0 = 1100\ \mathrm{kg\,m^{-3}}$, $\delta = 4.5 \times 10^{-6}\ \mathrm{m^2\,s^{-1}}$. For comparison with [6], the results reported below are for the simulation of 6000 time steps (the equivalent of one scan line computation) using a grid of size of 2064 $\times$ 2064.[4] The sound speeds within the medium were defined using a numerical phantom of a fetus [20]. This contained a small number of large homogeneous regions and other regions consisting of sub-wavelength scatterers. A snapshot showing the wave field as it propagates through the phantom is shown in Figure 2.

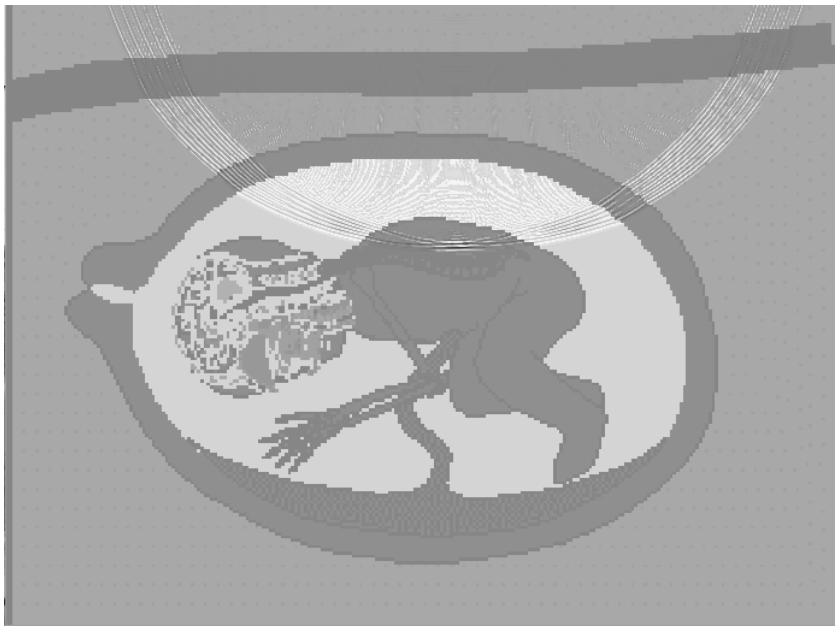

Fig. 2. Snapshot of an ultrasound wave propagating through a numerical phantom of a fetus.

Parallelisation (via use of multiple SPEs) of the code improves the performance of the code by approximately a factor of 5. The time required to perform this calculation when only one SPE is used is 284 seconds, compared to 57 seconds when all six of the available SPEs are used.

[3] The Courant-Friedrichs-Lewy (CFL) number is calculated by $(\Delta t\ c_{max})/\Delta x$, it measures the number of spatial grid points the fastest wave travels in one time step. Normally this will be a small fractional number and is dictated by stability constraints.

[4] [6] used a grid size of 2048 $\times$ 2048, however, as our grid dimensions needed to be a multiple of 24 we simulated on a slightly larger grid.

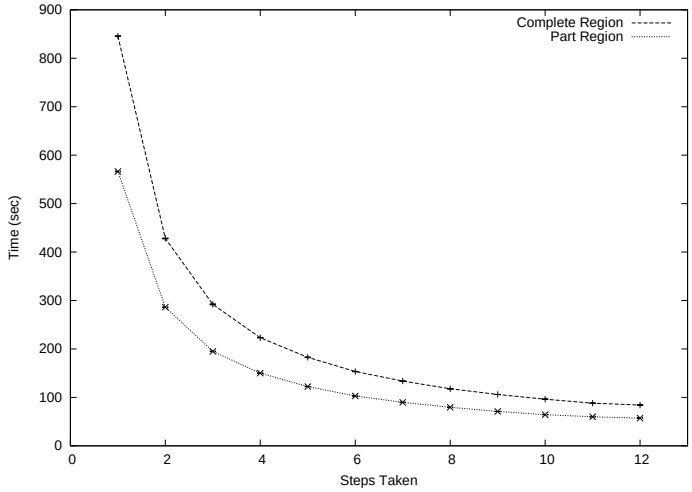

Fig. 3. Time required to compute a scan line (in seconds) vs number steps taken by a block.

Figure 3 shows how the time taken to compute one scan line changes as the number of steps between memory synchronisation increases. These results are calculated by repeating the execution 10 times for each measurement and reporting the average (the standard error on all these estimations is less than 0.9s). When the maximum amount of time steps (12) are calculated in between memory accesses, the time taken to calculate one scan line is 57 seconds. In [6], 192 scan lines (with the same grid size and number of timesteps) are computed in 55 minutes, approximately 17 seconds per scan line. Figure 3 also shows the improvement gained by restricting the simulation region, this shows an improvement of 32% over simulating the entire region over all time steps (for the maximum amount of time steps between memory synchronisation).

The performance of our code corresponds to an average main memory bandwidth usage of 7.2 GB/s, which is below the theoretically possible of 25.6 GB/s main memory bandwidth. Although, clearly there is room for improvement, such a figure is typical for applications like this.

6 Conclusion and Future Research

In this paper we described an implementation of nonlinear ultrasound simulation based on the Westervelt equation using the Cell Broadband Engine. The simulation time required is of the same order of magnitude as recent results achieved using GPGPUs [6] and avoids the need to use PC clusters, an approach used in other work [4]. The simulation of one scanline, which requires 57 seconds on the Cell, requires about 17 minutes when performed on a desktop PC with an Intel Core i7-2600K 3.40 GHz processor with

3 GB RAM. We have also shown the performance gained by doing multiple timesteps between memory sychronisation. It would be interesting to explore how this approach may be applied to other architectures such as a GPGPU.

A number of aspects of our implementation affect performance detrimentally. As mentioned above, it may be advantageous to allocate work to the SPEs using a work queue instead of allocating a fixed amount of work. This approach is used to do FDTD simulation in [11]. Performance may be also improved by communicating data and synronisation events between SPEs. This has the potential of saving DMA bandwidth to main memory and also it is known that synchronising with the PPE is highly latent [10]. Another source of performance improvement is to increase the size (and reduce the number of) DMA requests, because peak memory bandwidth usage cannot be achieved unless large requests are used. The present implementation uses many small requests, some of which are smaller than the minimum recommended size (128 bytes) [17, p. 455]. Thus, it may be possible to achieve substantial performance gains by rectifying these issues.

Performance gains may also be possible by generalising our novel idea that the simulation is limited to areas that can have an effect of the final ultrasound image. In order to keep our implementation simple, we perform this pruning (of the grid) rather crudely. For example, if a region is sufficiently far away from the transducer and the power of the ultrasound wave is sufficiently small, such a region may not require simulation calculations for the overall simulation to still be sufficiently accurate. It may be possible to, for example, adopt a less accurate FDTD scheme for those regions.

The use of spectral and k-space methods, which do part of their calculations in the frequency domain, such as those in [5, 21–23] has been shown to be effective. These methods provide high spatial accuracy [22, p. 917] and the Cell has already shown that it can efficiently perform the FFT [9, 10] which is required to implement these spectral and k-space methods. Thus it would be interesting to explore the effectiveness of these method on the Cell.

More realistic modelling may be achieved by substituting the thermoviscous absorption term in the Westervelt equation with a more general integro-differential operator that can account for power law absorption [24, 5]. It would be worth understanding and evaluating the exact performance costs of such modelling improvements.

Our promising results indicate that implementing a 3D ultrasound simulation on this hardware may be worthwhile. The implementation of absorbing boundary conditions (such as Berenger's PML adapted for ultrasound in [4]) would be necessary in order to accurately compare results with those gained from experimental data.

While the Cell processor is perhaps a non-standard computing platform (particularly in the ultrasound area), the push towards large-scale simulations requires both novel hardware and numerical approaches to make the calculations tractable. Moreover, emerging parallel architectures such as APUs or the Cell are likely to form an integral part of high performance computing clusters in the future. For example, a number of Cell clusters already exist, including the well known Roadrunner super computer.[5] Our

[5] The Roadrunner includes 12,240 PowerXCell 8i processor and was the world's fastest computer in 2008-2009; see http://www.lanl.gov/roadrunner/ .

research provides a good starting point for tackling much bigger ultrasound problems using a Cell cluster.

In summary, ultrasound simulation on the Cell can be performed much faster than using a standard single threaded desktop CPU and comparable to that of a GPGPU. Moreover, lessons learnt in our research are almost certainly transferable to any new architecture that emerges in the next decade.

Acknowledgements. We would like to thank the College of Engineering and Computer Science at The Australian National University for supporting this work through the summer scholar program. Bradley Treeby is supported by the Australian Research Council/Microsoft Linkage Project LP100100588.

References

1. Matte, G.M., Van Neer, P.L.M.J., Danilouchkine, M.G., Huijssen, J., Verweij, M.D., de Jong, N.: Optimization of a phased-array transducer for multiple harmonic imaging in medical applications: frequency and topology. IEEE Trans. Ultrason. Ferroelectr. Freq. Control **58**(3) (2011) 533–546
2. Huang, J., Holt, R.G., Cleveland, R.O., Roy, R.A.: Experimental validation of a tractable numerical model for focused ultrasound heating in flow-through tissue phantoms. J. Acoust. Soc. Am. **116**(4) (2004) 2451–2458
3. Wein, W., Brunke, S., Khamene, A., Callstrom, M.R., Navab, N.: Automatic CT-ultrasound registration for diagnostic imaging and image-guided intervention. **12**(5) (2008) 577–585
4. Pinton, G., Dahl, J., Rosenzweig, S., Trahey, G.: A heterogeneous nonlinear attenuating full-wave model of ultrasound. IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control **56**(3) (2009) 474–488
5. Treeby, B.E., Tumen, M., Cox, B.: Time domain simulation of harmonic ultrasound images and beam patterns in 3d using the k-space pseudospectral method. In: MICCAI 2011, Part I, LNCS 6891. (2011) 363—370
6. Karamalis, A., Wein, W., Navab, N.: Fast Ultrasound Image Simulation using the Westervelt Equation. Medical Image Computing and Computer-Assisted Intervention (2010) 243–250
7. Varray, F., Cachard, C., Ramalli, A., Tortoli, P., Basset, O.: Simulation of ultrasound nonlinear propagation on GPU using a generalized angular spectrum method. EURASIP Journal on Image and Video Processing **2011** (2011) 17
8. Bader, D., Agarwal, V., Madduri, K., Kang, S.: High performance combinatorial algorithm design on the Cell Broadband Engine processor. Parallel Computing **33**(10-11) (2007) 720–740
9. Chow, A., Fossum, G., Brokenshire, D.: A programming example: Large FFT on the Cell Broadband Engine [Online; accessed 24 January 2012].
10. Bader, D., Agarwal, V.: FFTC: Fastest Fourier transform for the IBM Cell Broadband Engine. High Performance Computing (2007) 172–184
11. Xu, M., Thulasiraman, P.: Parallel Algorithm Design and Performance Evaluation of FDTD on 3 Different Architectures: Cluster, Homogeneous Multicore and Cell/BE. In: 10th IEEE International Conference on High Performance Computing and Communications, IEEE (2008) 174–181
12. Li, B., Jin, H., Shao, Z.: Two-Level Parallel Implementation of FDTD Algorithm on CBE. In: IEEE International Conference on Networking, Sensing and Control, IEEE (2008) 1812–1817

13. Hamilton, M.F., Blackstock, D.T., eds.: Nonlinear Acoustics. Acoustical Society of America, Melville (2008)
14. Fornberg, B.: Generation of Finite Difference Formulas on Arbitrarily Spaced Grids. Mathematics of Computation **51**(184) (1988) 699—706
15. Hallaj, I., Cleveland, R.: FDTD simulation of finite-amplitude pressure and temperature fields for biomedical ultrasound. Acoustical Society of America **105** (1999) 7–12
16. Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., Shippy, D.: Introduction to the Cell multiprocessor. IBM Journal of Research and Development **49**(4.5) (2005) 589–604
17. IBM: Cell Broadband Engine Programming Handbook (2008) [Online; accessed 24 January 2012].
18. Koranne, S.: Practical computing on the Cell Broadband Engine. Springer, New York London (2009)
19. IBM: C/C++ Language Extensions for Cell Broadband Engine Architecture. Volume 2.4. (March 2007)
20. Jensen, J.A., Munk, P.: Computer phantoms for simulating ultrasound B-mode and cfm images. In: 23rd Acoustical Imaging Symposium. (1997) 75–80
21. Daoud, M., Lacefield, J.: Parallel three-dimensional simulation of ultrasound imaging. In: 22nd International Symposium on High Performance Computing Systems and Applications, IEEE (2008) 146–152
22. Liu, Q.: Large-scale simulations of electromagnetic and acoustic measurements using the pseudospectral time-domain (PSTD) algorithm. IEEE Transactions on Geoscience and Remote Sensing **37**(2) (1999) 917–926
23. Tabei, M., Mast, T., Waag, R.: A k-space method for coupled first-order acoustic propagation equations. The Journal of the Acoustical Society of America **111** (2002) 53–63
24. Purrington, R.D., Norton, G.V.: A numerical comparison of the westervelt equation with viscous attenuation and a causal propagation operator. Mathematics and Computers in Simulation **82**(7) (2012) 1287 – 1297