Acceleration of GPU-based ultrasound simulation via data compression

Andrew A. Haigh, Eric C. McCreath
Research School of Computer Science,
The Australian National University,
Canberra, Australia
Email: andrew.haigh@anu.edu.au, eric.mccreath@anu.edu.au

***Abstract*—The realistic simulation of ultrasound wave propagation is computationally intensive. The large size of the grid and low degree of reuse of data means that it places a great demand on memory bandwidth. Graphics Processing Units (GPUs) have attracted attention for performing scientific calculations due to their potential for efficiently performing large numbers of floating point computations. However, many applications may be limited by memory bandwidth, especially for data sets whose size is larger than that of the GPU platform. This problem is only partially mitigated by applying the standard technique of breaking the grid into regions and overlapping the computation of one region with the host-device memory transfer of another.**

In this paper, we implement a memory-bound GPU-based ultrasound simulation and evaluate the use of a technique for improving performance by compressing the data into a fixed-point representation that reduces the time required for inter-host-device transfers. We demonstrate a speedup of 1.5 times on a simulation where the data is broken into regions that must be copied back and forth between the CPU and GPU. We develop a model that can be used to determine the amount of temporal blocking required to achieve near optimal performance, without extensive experimentation. This technique may also be applied to GPU-based scientific simulations in other domains such as computational fluid dynamics and electromagnetic wave simulation.

***Keywords*—GPGPU, Data compression, Parallel architectures, Memory architecture, Nonlinear acoustics**

I. Introduction

The problem of efficiently simulating the propagation of ultrasound waves has many important applications, such as designing ultrasound imaging systems and high-intensity therapeutic treatment [1], [2]. This simulation may be done by deriving and then solving numerically a system of stencil-like finite-difference equations [1], [3], [4], [5]. A variety of different nonlinear effects such as power law absorption and thermoviscous attenuation can be modelled with this approach. GPUs are particularly of interest for this problem as they have tremendous potential for performing scientific calculations both quickly and energy efficiently [6]. This is due to their inherently parallel architecture, containing hundreds of cores able to execute calculations in parallel [7].

Realistic finite-difference ultrasound simulations are very computationally intensive as achieving accuracies of practical interest require at least 10 grid points per wavelength [8]. Consequently, a typical simulation may require a grid with a side length of hundreds or thousands of grid points. This places a great demand on memory, especially in the case of 3D simulations.

It is well known that stencil computations are frequently memory bound [9]. As a consequence, one of the major difficulties in achieving good performance on a GPU is overcoming the memory bottlenecks inherent in the GPU architecture. Here we focus our attention on the issue of performing 2D simulations where the data is too large to fit completely on the device memory present on the GPU. This necessitates subdividing the problem into smaller pieces, and the bottleneck is the rate at which data can be transferred between the CPU and GPU. By compressing the data before it is transferred, the transfer rate is effectively increased, at the cost of extra computation (to perform the compression/decompression) and accuracy.

An additional benefit in compressing the data is that the GPU can operate on larger data sets in between communication with the CPU. This has the effect of reducing the relative amount of superfluous transfers/computation that has to be done in order to accommodate the surrounding 'ghost' regions needed to simulate the region of interest.

We demonstrate that for data sets too large to fit in the GPU memory, overall performance is improved by the use of a lossy compression scheme. This is true even when we perform double buffering and temporal blocking to completely overlap the computation and memory transfers.

II. Related Work

A. FDTD for ultrasound simulation

The finite-difference time domain (FDTD) technique was introduced by Yee [10] for the problem of numerical simulation of electromagnetic wavefields. It involves discretising the region into a set of grid points and calculating an approximation to the solution at each grid point. The simulation is stepped forward in time using a stencil-like calculation with weights chosen to minimise the local truncation error [11].

The method has been applied to the problem of ultrasound simulation by many researchers [1], [3], [4], [5]. In addition, there has been work on performing these simulations on a variety of different architectures, including GPUs [12], [13], and the Cell broadband engine [14].

B. Compression of floating-point data

There have been a number of attempts to use data compression to improve the performance of bandwidth-limited scientific applications. The simplest of these are compression algorithms that can be applied to each piece of data independently, which allows for efficient parallelisation [15], [16].

Due to the fact that GPU hardware was traditionally shaped by the requirements of computer graphics applications [7], much work on compression algorithms that run on GPUs has been devoted to compressing the representation of graphical structures such as meshes in order to facilitate applications such as efficient remote rendering [17]. Similar techniques can be used to compress the storage of sparse matrices [18].

In general, compression of floating point data is significantly different to the compression of discrete data, as in order to achieve a good compression ratio the compressor must take account of the specifics of the representation of the floating-point data. This is because of the different interpretation of the bits representing the sign, significand and exponent. One solution to this problem is to rewrite all the floating point data in a fixed point format with an extremely large number of bits, to ensure no information is lost, then apply standard compression techniques [19].

A common approach is to use a scheme that predicts each value then encodes the difference between the predicted and actual value [20], [21], [22], which may be done efficiently if the residual is usually small. Among the most significant of these is a lossless compression scheme called FPC which uses a polynomial interpolation-like prediction [23]. A variant of this algorithm called GFC has been adapted to run efficiently on GPUs [24].

Data that is the solution of a hyperbolic PDE, as in our case, is likely to have a sparse representation under an alternative basis. This can be exploited by techniques such as a discrete Fourier or wavelet transform. Such techniques have been successfully applied to the compression of floating point scientific data [25], [26].

III. Background

A. Ultrasound simulation

For this work we solve, in two spatial dimensions, the Westervelt equation, a nonlinear partial differential equation (PDE) that models wave propagation in addition to the effects of thermal attenuation and cumulative nonlinear effects. It is given by [12]

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

where p is the acoustic pressure (Pa), c is the propagation speed (ms^{-1}), ρ_0 is the ambient density (kgm^{-3}), δ is the diffusivity of sound ($\mathrm{m}^2\mathrm{s}^{-1}$), and β is the coefficient of nonlinearity. A derivation can be found in [27].

We discretise with an equally spaced grid with points (x_i, y_j) where $x_i = i\Delta x$ and $y_j = j\Delta y$, and equally spaced timesteps $t_n = n\Delta t$ such that we approximate $p(x_i, y_j, t_n) \approx p^n_{i,j}$. We use the finite differences

$$\frac{\partial^2 p}{\partial t^2} \approx \frac{p^{n+1}_{i,j} - 2p^n_{i,j} + p^{n-1}_{i,j}}{\Delta t^2}, \tag{2}$$

$$\frac{\partial^3 p}{\partial t^3} \approx \frac{3p^{n+1}_{i,j} - 10p^n_{i,j} + 12p^{n-1}_{i,j} - 6p^{n-2}_{i,j} + p^{n-3}_{i,j}}{\Delta t^3}, \tag{3}$$

and expand $\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}$ and $\frac{\partial^2 p^2}{\partial t^2} = 2\left(\left(\frac{\partial p}{\partial t}\right)^2 + p\frac{\partial^2 p}{\partial t^2}\right)$ where

$$\frac{\partial p}{\partial t} \approx \frac{1}{\Delta t}\left(\frac{11}{6}p^n_{i,j} - 3p^{n-1}_{i,j} + \frac{3}{2}p^{n-2}_{i,j} - \frac{1}{3}p^{n-3}_{i,j}\right), \tag{4}$$

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

$$\frac{\partial^2 p}{\partial x^2} \approx \frac{-p^n_{i+2,j} + 16p^n_{i+1,j} - 30p^n_{i,j} + 16p^n_{i-1,j} - p^n_{i-2,j}}{12\Delta x^2}, \tag{6}$$

with the corresponding scheme for $\frac{\partial^2 p}{\partial y^2}$.

This leads to an explicit scheme which can be solved for $p^{n+1}_{i,j}$ in terms of the pressure at the same grid point in the previous 4 timesteps and values $p^n_{i+k,j}, p^n_{i,j+k}$ for $k = -2, -1, 1, 2$. To ensure stability, Δx and Δt should be chosen such that the Courant-Friedrichs-Lewy (CFL) value $\frac{c_{\max}\Delta t}{\Delta x} \ll 1$. A similar scheme is used in [28] and in cylindrical coordinates in [1].

B. Compression

In this work we use a simple lossy scheme to compress the single-precision floating point data by converting it into a fixed point representation. This entails representing all values as a sign and significand, with the exponent implicit by the fact that it is the same for all values.

The reason for choosing this compression scheme is that the values handled by the simulation occupy a relative small range of orders of magnitude, so storing the exponent separately for each data point is largely redundant. In addition, the actual compression can be carried out relatively efficiently. It can be implemented without using a large number of bit-manipulation instructions, which are unfavourable for GPU execution. This makes it a good candidate to test whether compression is a practical technique for improving performance.

Data stored in a floating point format consists of a sign, exponent and significand. We exploit the fact that all floating-point values in the range $[2^n, 2^{n+1})$ have the same exponent, hence, if the values are constrained to this range, the exponent does not need to be stored. Assuming the data is contained in the range $(-2^n, 2^n)$ compression is performed by either adding or subtracting 2^n depending on the sign so that the data lies in the range $(-2^{n+1}, -2^n) \cup [2^n, 2^{n+1})$ and, consequently, the exponent can be discarded. Decompression involves performing the same operations in reverse.

We consider two schemes: retaining either the 15 or 23 most significant bits of the significand (in addition to the sign bit). These correspond to compressing 2 (single precision) values into 4 bytes or 4 values into 12 bytes. We refer to these as the 2 byte and 3 byte compression schemes, respectively. Note that even though the 3 byte compression utilises the

```
#define EXP 2
#define HIGH 23
#define DELTA (1<<EXP)
void compress(float* in, char* out_c) {
      unsigned int *out =
            (unsigned int*)out_c;
      unsigned int sign =
            (*(unsigned int*)in) & (1<<31);
      float v = *in + (sign ? -DELTA : DELTA);
      *out = ((*(unsigned int*)&v) &
          ((1<<HIGH)-1)) + (sign?(1<<HIGH):0);
}
```

Fig. 1. Code for compressing single precision floating point value into 3 bytes for data in range $(-2^n, 2^n)$ with $n = 2$

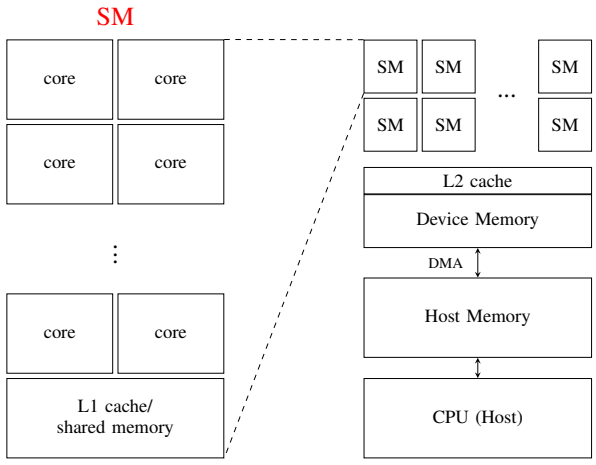

Fig. 2. Schematic of GPU architecture

entire significand, it is not a lossless scheme because lower-significance bits are lost when adding (or subtracting) 2^n to the initial value.

Fig. 1 gives the code to take a standard single-precision floating-point stored in 4 bytes and compress it into 3 bytes. The implementations for decompression and for the 2 byte scheme are analogous.

IV. GPU ARCHITECTURE AND PROGRAMMING MODEL

NVIDIA GPUs are subdivided logically at a number of levels, as shown in Fig. 2. At the highest level, a single GPU is divided into a number of streaming multiprocessors (SMs). Each streaming multiprocessor consists of many cores that can execute threads in parallel [7]. As threads are grouped into warps that execute in lockstep, GPUs can be considered a kind of massively-SIMD architecture. Each SM switches frequently between warps to hide high-latency instructions. The exact specifics of the hardware such as the number of SMs per GPU, cores per SM, threads per warp and amounts of memory available vary depending on the exact GPU and its generation.

Memory consists of a similar hierarchy: registers private to an individual thread, shared memory available to an entire thread block, and global memory available to all threads. Global memory is a separate address space to any existing CPU memory and any copies must be managed by and initiated from the CPU.

The user is exposed to this hardware via the NVIDIA CUDA programming model, an extension to C that allows designated code (*kernels*) to be run on the GPU. Synchronisation during kernels is only allowed between threads in the same block, and global synchronisation is only generally possible by finishing a kernel and returning control to the CPU, at which point register and shared memory contents are flushed.

V. APPROACH

We implemented a simulation that carries out the computation from Section III-A. The calculation must be divided into disjoint parts since we are interested in data sets that are too large to fit in the GPU global memory. To do this we make use of the ability of the GPU hardware and CUDA programming model that allows the user to perform memory transfers and computations (in the form of kernels) simultaneously. The grid is divided conceptually into slices that each consist of a contiguous subset of entire rows. We make use of double-buffering so the memory transfer necessary to simulate one slice can be done whilst the actual simulation of another slice is being performed.

To do this we must declare on the host enough space to store 10 copies of the entire *uncompressed* simulation grid: 4 previous timesteps, space for the next timestep (to avoid dependency issues), and the same again in order to facilitate double buffering. The host memory must be declared page-locked in order to issue asynchronous copies. Similarly, on the GPU we declare enough space for 10 copies of the largest slice in compressed form. The grids are used in a cyclic fashion i.e. when storing a new set of values from the next iteration, it is stored in place of the oldest values, which are no longer required.

Algorithm 1 Algorithm for performing the out-of-core simulation with double-buffering and temporal block size t

1: Allocate memory for grids on CPU and fill with initial conditions
2: Perform initial compression (if any) of data on CPU
3: Choose row boundaries $y_i\,(i = 0, 1...n)$ at which the calculation shall be divided into parts
4: Issue copy contents of rows $[y_0, y_1)$ + t-row ghost region at timesteps $[-4, 0)$ to GPU memory
5: **for** $i = 0, t, 2t, ...T - t$ **do**
6: **for** $j = 0 \to n - 1$ **do**
7: Launch sequence of kernels that calculates contents of rows $[y_j, y_{j+1})$ for timesteps $[i, i + t)$, decompressing and compressing as required
8: Issue copy of contents of rows $[y_{j+1}, y_{j+2})$ + t-row ghost region at timesteps $[i-4, i)$ to GPU memory
9: Issue copy of contents of rows $[y_j, y_{j+1})$ at timesteps $[i + t - 4, i + t)$ back to CPU memory
10: **end for**
11: **end for**
12: Perform final decompression (if any) of data on CPU

Algorithm 1 gives the steps for performing this procedure including the double buffering and handling of multiple timesteps between copies. For small temporal block size $t < 4$,

our implementation takes special consideration (not described in Algorithm 1) to avoid redundant copies by only copying back the results of the previous t (instead of 4) timesteps.

Our implementation divides the grid into slices by choosing $y_i\,(i = 0, 1, ...n)$ such that $0 = y_0 < y_1 < ... < y_n = Y$ and the number of rows in the largest slice $\max_i\,(y_{i+1} - y_i)$ is small enough to fit entirely on the device memory. The number of slices n is chosen to be as small as possible within this memory constraint as described below in section VI. In many cases it will not be possible to divide the grid evenly but our algorithm aims to balance the workload (number of rows in each slice) as fairly as possible. In addition, we avoid a complicated array boundary check by ensuring that the number of rows in the last slice $[y_{n-1}, y_n)$ is a multiple of the block height.

For the actual kernel that performs the computation there are many factors (such as thread block size) that affect the performance. We have empirically tested some common optimisations in order to achieve good performance. These are likely to be heavily device-specific. Since we have separate kernels for operating on the compressed and uncompressed data, we have tried to optimise each in a balanced manner.

VI. Modelling performance

We give a simple model that estimates the performance of the code as a function of a number of important parameters. We are particularly interested in the behaviour depending on t, the number of timesteps performed in each temporal block.

For a grid of size $X \times Y$ and a system with M bytes of memory available on the GPU, if we compress by a factor of λ (where $\lambda = 0.5$ for the 2 byte compression scheme and $\lambda = 0.75$ for the 3 byte compression scheme), then we can simulate t timesteps without any intermediate CPU-GPU transfers on a slice of size $X \times y$ iff

$$40\lambda X(y + 4t) \leq M \tag{7}$$

This is due to the fact that each data point is 4λ bytes and we store 10 grids of the desired size. In addition, for each extra timestep to perform, we need 4 extra rows (2 above and below), since the simulatable region shrinks after each timestep because the adjacent pressures are not available. To minimise the total amount of computation and transfer, for a fixed t, it is desirable to choose the largest possible y, which is given in most cases by $y(t) = \left\lfloor \frac{M}{40X\lambda} - 4t \right\rfloor$. This does not apply when it is necessary to allocate an extra row due to the fact that the number of rows is not evenly divisible by y, but our model does not consider this.

Using T to denote the total number of timesteps, then the entire simulation is divided up into $\left\lceil \frac{T}{t} \right\rceil$ blocks of timesteps, each of which (except possibly the last) consists of t timesteps on $\left\lceil \frac{Y}{y(t)} \right\rceil$ independent slices of data. For a single slice, each block of t timesteps involves filling 4 of the 10 grids, containing the pressures from the previous 4 timesteps, on the GPU with data that is copied from the CPU. Similarly, it requires reading back the data in the *non-ghost region* ($y(t)$ of the $y(t) + 4t$ rows) from the final $\min(t, 4)$ timesteps. Assuming the entire M available bytes are used the total amount of host-to-device data transfers is therefore $\frac{4}{10}M$ bytes, and the total amount of device-to-host transfers is $\frac{\min(t,4)}{10}\frac{y(t)}{y(t)+4t}M$ bytes.

In addition, let R be the CPU-GPU data transfer rate in bytes/second (assuming for simplicity the same rate for both host-to-device and device-to-host transfers). Let C_λ be the rate of performing the actual simulation in seconds per grid point per timestep, including the cost of and varying depending on the amount of compression/decompression. These values can be estimated both theoretically and using data from NVIDIA Visual Profiler.

Since we can shrink the ghost region in a block after each timestep, the total grid points to simulate t steps is the number of columns X multiplied by $y + (y+4) + ... + (y+4(t-1)) = yt + 4\frac{t(t-1)}{2}$. Finally, assuming the computation and transfer are totally overlapped, the total runtime is given roughly by the function of t,

$$\left\lceil \frac{T}{t} \right\rceil \left\lceil \frac{Y}{y(t)} \right\rceil \max\left(\overbrace{\left(\frac{4}{10} + \frac{\min(t,4)}{10}\frac{y(t)}{y(t)+4t} \right)\frac{M}{R}}^{\text{transfer}}, \overbrace{C_\lambda X\{y(t)t + 2t(t-1)\}}^{\text{computation}} \right) \tag{8}$$

This model illustrates the tradeoff between doing many timesteps and saving memory transfers and doing few timesteps and having smaller ghost regions.

VII. Results

We test our implementation on a Fermi architecture GPU, the NVIDIA GTX 580. It is connected to a server running Ubuntu 12.04 server edition via a PCI-e 2.0 x16 bus. In our implementation all memory allocations on the host are declared page-locked so that the corresponding data will reside in the 47 GBytes of DDR3 DRAM available on our system. Though the PCI-e 2.0 x16 bus has a peak transfer rate of 16 GBytes/s, our CPU-GPU transfers achieve around 5.2 GBytes/s, as measured by the CUDA Visual Profiler, which is around the known limit when using `cudaMemcpy` [29].

The GTX 580 has 512 CUDA cores and approximately 1.5 GBytes of global device memory. We compile the CUDA-specific code with the NVCC compiler from CUDA toolkit 5.0 targeted to compute capability 2.0 (the highest supported by the GTX 580) and the remainder with GNU C++ compiler version 4.6.4.

Each iteration consists of applying the iteration from Section III-A at each grid point. When simulating points on the boundary of the grid, we assume that the solution takes the value 0 at all points outside the simulated region.

We have used the parameter values $\Delta t = 10^{-9}$ (s), $\Delta x = 10^{-5}$ (m) and $c = 1500$ (ms^{-1}), giving a stable CFL value of 0.15. For the nonlinear parameters we have used $\delta = 4.5 \times 10^{-3}$ (m^{2}s^{-1}), $\beta = 6$ and $\rho_0 = 1100$ (kgm^{-3}), the same as Karamalis et al. [12].

We have implemented an equivalent version of the simulation in C++ using double precision calculations. This was done for the purpose of checking correctness of the GPU version and to quantify the error introduced by each compression scheme. Our GPU (using uncompressed single precision data) and CPU

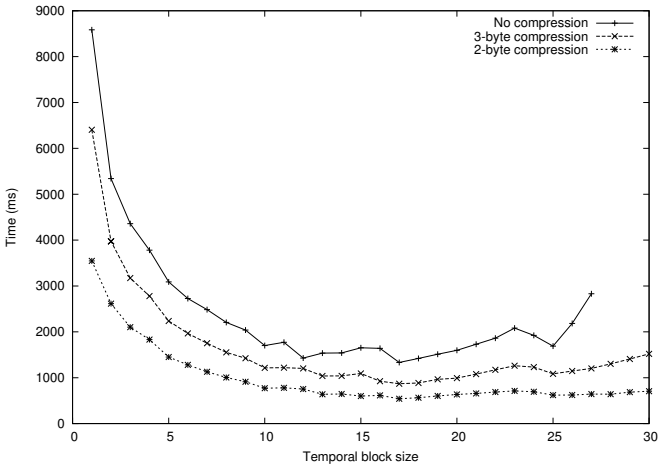

Fig. 3. Runtime as a function of temporal block size (t), for each compression scheme, for a 40960×1024 grid restricted to 200 MBytes memory usage

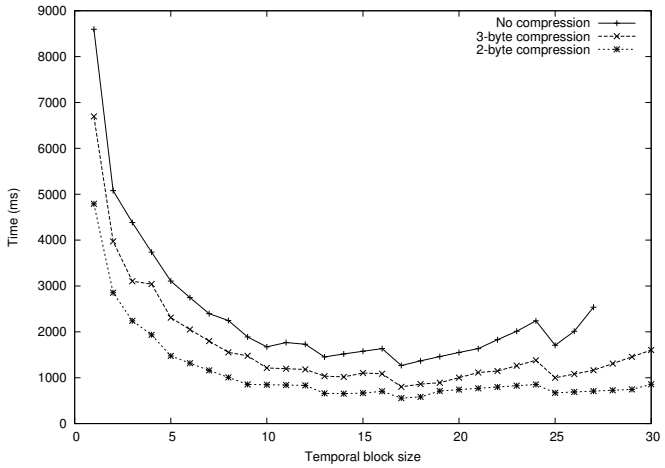

Fig. 4. *Predicted* runtime as a function of temporal block size (t), for each compression scheme, for a 40960×1024 grid restricted to 200 MBytes memory usage

implementations are in agreement to the expected level of precision, which suggests that our GPU implementation is correct.

For performance results, we perform simulations with 50 timesteps. Each test is repeated 5 times and we report the minimum time as measured by the CUDA event API. As expected this differs only very slightly from the average time in almost all test cases, as our implementation is entirely deterministic and so any variation in runtime will be solely due to external factors. The only observed fluctuation seems to be due to the creation/destruction of CUDA device contexts.

For the initial conditions, we select a subset of (1000) grid cells at random and initiate them with a square-wave pulse i.e. set the pressure to be 1 for the first 4 timesteps. Correspondingly our compression scheme assumes all pressure values to lie in the range $(-4, 4)$. The range of values handled by a simulation is highly domain and application specific but we feel this to be a representative scenario.

All results ignore the cost of the initial compression and final decompression that must be performed on the CPU. We have made no effort to optimise these calculations e.g. by using multiple threads. In addition, for a fixed grid size, this cost is constant (for a 64512×1024 grid, it is around 4 seconds, similar to around 100 timesteps of simulation), and is effectively amortised when performing a simulation with a large number of timesteps.

A. Performance

First we describe the execution time for a simulation where we artificially limit the amount of GPU device memory used to 200 MBytes. This allows a comparison with the case where the entire simulation can be fit onto the device and carried out without any intermediate transfers, which illustrates quantitatively the cost of splitting the computation into separate parts.

We use a grid of 40960 columns and 1024 rows. This is not a particularly large simulation compared to many 3D simulations since the number of grid points is less than 400^3.

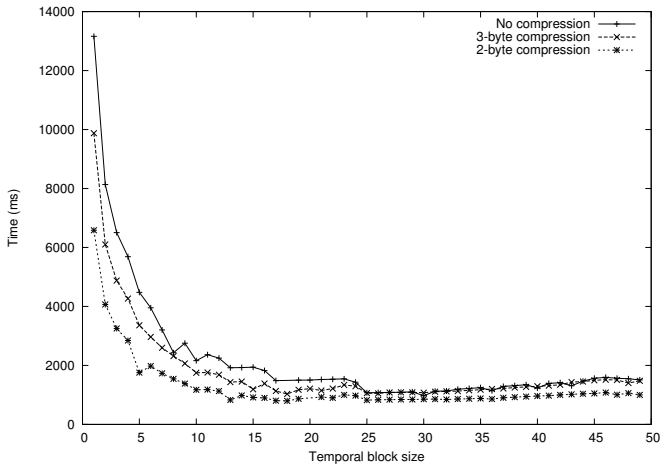

Fig. 5. Runtime as a function of temporal block size (t), for each compression scheme, for a 64512×1024 grid restricted to 1 GByte memory usage

Since we divide the problem into slices of entire rows, the total number of timesteps and height do not affect the runtime except by an approximately constant multiple, and so the relative speedup we achieve will be roughly the same for a simulation of the same width but with many more rows and timesteps (e.g. a 40960^2 grid).

Fig. 3 gives the execution time for this simulation as a function of t, the maximum number of timesteps performed in between transferring results back to the host. Table I summarises the results assuming the optimal choice of temporal block size t. For this small simulation with a restricted amount of memory, both the 2 byte and 3 byte compression schemes allow improved performance. The 2 byte compression scheme gives a speed up more than 2 times while the 3 byte compression scheme gives a speedup of approximately 1.5 times. Our model can explain why the speedups for this test case are greater than the ratio to which the data is compressed. This is because compression gives the double benefit of not only reducing transfer time for memory bound simulations, but

it allows larger (in terms of number of rows) slices to be fit in the device memory, since the data is only compressed/decompressed by the kernel as required. This factor reduces the relative amount of data required to transfer the ghost regions surrounding each slice.

We note that as a rule of thumb for fixed t, for memory-bound cases the runtime should be roughly inversely proportional to the compression performed. Indeed this is observed when $t = 1$ as the uncompressed, 3 byte and 2 byte simulations require around 8500 ms, 6500 ms, and 3700 ms respectively. This rule begins to break down for larger t as the cost of computation and amount of memory transfer/computation required to handle the ghost regions grows differently depending on the compression scheme used.

Ignoring the artificial memory constraint, the simulation when fit entirely on the device, runs in around 400 ms, ignoring the cost of the initial and final memory transfers. Table I details how this compares to the case when only 200 MBytes is used. It is clear that the memory constraint, even with optimal choice of t, so that transfers are overlapped by computation, causes significant slowdown. The uncompressed version requires around 1300 ms, more than three times as slow. This demonstrates quantitatively how compression pushes the runtime toward the performance bound of the simulation performed entirely on the device.

Fig. 4 gives the predicted runtime for the same simulations using the model introduced in Section VI. The data transfer rate R is estimated to be 5.2 GBytes/s using information from the NVIDIA Visual Profiler. The time required to simulate a single grid point C_λ is estimated from a number of small benchmark test cases carried out entirely on the device memory. We use the values $C_1 = 1.95 \times 10^{-7}$ ms, $C_{0.75} = 2.61 \times 10^{-7}$ ms, $C_{0.5} = 2.13 \times 10^{-7}$ ms for no compression, 3 byte compression and 2 byte compression schemes respectively.

It is apparent that the predicted (Fig. 4) and actual (Fig. 3) runtimes match qualitatively. For example, the model is able to predict the sharp increase in runtime in the simulation with uncompressed data as $t \to 30$, as the ghost region becomes so large that the expense of transferring its contents/simulating it begins to become a substantial fraction of the total computational cost.

It is clear from both the experimental results and the predictive model that the runtime as a function of the temporal block size is not monotonic (decreasing then increasing). There are some parameter choices such that increasing the temporal block size by 1 actually causes a decrease in performance. This is due to the fact that performing the extra timestep in between communication requires the storage of 4 additional rows of data on the GPU. The limited amount of memory may force an increase in the total number of pieces into which the entire grid must be divided to perform the calculation. This increases the total amount of work (and hence runtime) since the total number of grid points involved in the ghost regions around each piece is increased.

This sharp change in runtime between some parameter values is more pronounced in the predictive model than the experimental results. This is due to the fact that the model assumes a worst case scenario in terms of load balancing, i.e.

TABLE I. TIME IN MILLISECONDS FOR TWO SIMULATIONS FOR OPTIMAL SIZE OF TEMPORAL BLOCK, WITH DIFFERENT COMPRESSION SCHEMES

	Runtime (ms)	Speedup	Runtime (ms)	Speedup
Columns	40960		64512	
Available memory	200 MBytes		1 GBytes	
No compression	1333.24	1.000	970.37	1.000
3 byte compression	869.25	1.534	1032.37	0.940
2 byte compression	542.02	2.460	796.53	1.218
Entirely on device	399.52	3.337	n/a	

completely uniform runtimes and a fixed schedule. In practice random effects such as memory bus contention combined with warp scheduling help to mitigate this factor. In addition, we note that the slight mismatch between the location of the sharp changes between the two graphs is due to the fact that our simple model may sometimes incorrectly determine the number of pieces into which the grid must be subdivided. This includes the aforementioned complicated case such as the allocating of an extra row in case the grid is not divisible into equal pieces.

Intuitively we expect that because for a single slice computation increases and memory transfers decrease as a function of the temporal block size, the optimal number of timesteps t is such that the memory transfer overlaps as close as possible with the computation. Our model suggests that for this size grid and memory constraint, when using 2 byte compression this occurs when $t = 15$, when using 3 byte compression it occurs when $t = 19$ and when operating on uncompressed data the simulation is always memory bound. Fig. 4 indicates that for 2 and 3 byte compression schemes these values of t give very close to optimal performance. This suggests that our simple model is a very useful tool for getting good performance in simulations of other sizes without extensive experimentation.

Fig. 5 gives the results for a larger test case using 1 GByte of device memory, which is the majority of the 1.5 GBytes available on the GTX 580. This simulation uses a grid of 1024 rows and 64512 columns, and is otherwise the same as the smaller case described above. Assuming the choice of optimal temporal block size t, the 2 byte compression scheme provides a speedup of around 1.2 times (see Table I). In addition, we note that for almost all choices of the temporal block size t, the 3 byte compression scheme outperforms the uncompressed simulation. However, for optimal choice of t, the simulation on uncompressed data performs better. We believe that this is due to the fact that for this particular choice of t, the simulation happens to divide conveniently into a number of pieces in a way that slightly favours operating on uncompressed data.

This result combined with the small test case indicate that compression is most beneficial when the problem size is large relative to the amount of memory available. We note that the relatively poor results on the large test case do not indicate that our approach does not scale with problem size because the important factor is not the absolute problem size but how it compares to the available memory.

We can use our model to quantify this assertion by using it to predict the grid size required for compression to beneficial. With this we can predict (simply by testing many different values) the minimum value of X (the number of columns in the grid) required when M bytes of memory are available for a given compression scheme to be beneficial. The number

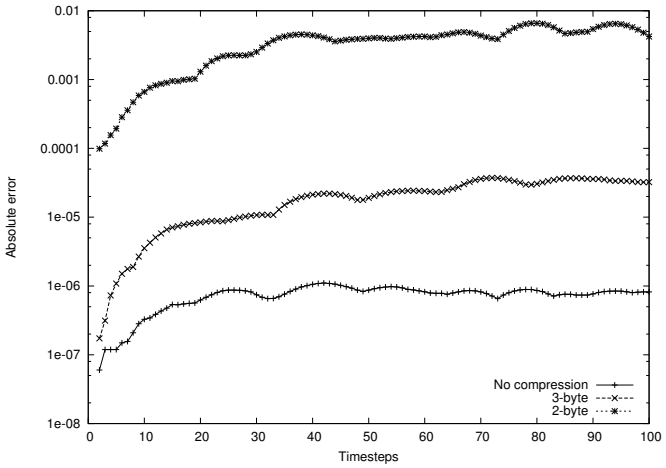

Fig. 6. Maximum absolute error over all grid points in a sample simulation as a function of number of timesteps and compression scheme

of rows is not important because we can break the grid into more pieces. When 1 GByte is available, 2 byte compression becomes beneficial at around $X = 58000$ and 3 byte compression at around $X = 90000$. This concurs with the result we have observed for this test case. For the small test case with only 200 MBytes of memory available, 2 byte compression becomes beneficial at around $X = 16000$ and 3 byte at around $X = 24000$, which suggests speedups in both cases, which were indeed observed.

B. Error

Here we quantify empirically the error introduced by both the 2 byte and 3 byte compression schemes. We do this by comparing the results for simulations of a variable number of timesteps to our serial double precision described above. In addition, we can compare the error to that introduced by single precision by performing the simulation in using our baseline GPU implementation.

The results are reported for a simulation on a grid of size 1024^2 with a single square-wave pulse of duration 4 timesteps propagating from the center of the grid. All other simulation parameters are the same as for the performance results detailed above.

Fig. 6 shows the maximum absolute error over all grid points as a function of the number of timesteps simulated, when using single precision, 2 byte compression and 3 byte compression, compared to double precision. Absolute error is used as a metric instead of relative error because we believe that the wave dissipation makes the relative error appear overly pessimistic. We conclude that for this domain, the 3 byte compression scheme is useful for performing at least qualitatively accurate simulations. Unfortunately, in this domain, the error in the 2 byte compression scheme seems to accumulate too much for this scheme to be useful.

The absolute error seems initially to grows approximately linearly as a function of the number of times compression is applied. Notably it is apparent that the error is non-negligible even for the uncompressed (single precision) simulation. In this sense we can interpret the 2 byte and 3 byte compression schemes as an extension of the transition from double to single precision, that illustrate the trade off between computation time and simulation accuracy. In terms of performance we expect a memory-bound GPU-based double precision simulation would be approximately twice as slow as the single precision version, as twice as much data must be transferred.

The pattern in the error as a function of timestep plotted in Fig. 6 is worth noting. We conjecture that the periodicity seen here is related to the wavelength of the wave that is being simulated, as they are both approximately in the range of 10 to 15 timesteps.

VIII. Conclusion and Future Research

In this paper we have investigated the performance benefits of using data compression to improve effective memory bandwidth on a stencil-like ultrasound simulation. We have demonstrated a speedup of 1.5 times on a simulation that compresses single-precision floating point values into 3 bytes, whilst maintaining at least qualitative accuracy in the final result. We believe that this technique and the further improvements described below will be useful for a wide range of other applications including electromagnetic wave propagation, seismological simulations and computational fluid dynamics.

Our results show that in principle it is possible that by dealing with a data set in compressed form including during communication across a memory bottleneck, overall performance can be improved. This is true even when the memory transfer can be entirely hidden by using temporal blocking. Our approach is likely to apply similarly to hardware utilising similar memory models, such as those having separate host and device address spaces, including a wider range of GPUs supporting the OpenCL standard.

We plan to investigate the effect of more sophisticated compression schemes, such as wavelet-type compression algorithms, in order to achieve good performance combined with high accuracy. Due to the large-scale homogeneities in many pressure fields of interest, it should be possible to achieve very high compression with minimal loss in accuracy. We also plan to move to 3D simulations where the amount of memory available becomes a more severe limitation.

In the future it is likely that CPUs and GPUs and their address spaces will become more tightly integrated. However, this technique is equally applicable to the problem of performing simulations distributed across multiple GPUs. Since it applies broadly to problems where the computation is divided into many independent parts, it will work equally if the parts are divided between multiple GPUs instead of being performed sequentially on a single GPU.

Acknowledgement

The authors would like to thank Anish Varghese for his helpful comments and suggestions.

References

[1] I. Hallaj and R. Cleveland, "FDTD simulation of finite-amplitude pressure and temperature fields for biomedical ultrasound," *Acoustical Society of America*, vol. 105, no. 5, pp. 7–12, 1999.

[2] B. E. Treeby, J. Jaros, A. P. Rendell, and B. Cox, "Modeling nonlinear ultrasound propagation in heterogeneous media with power law absorption using a *k*-space pseudospectral method," *The Journal of the Acoustical Society of America*, vol. 131, p. 4324, 2012.

[3] T. D. Mast, L. M. Hinkelman, M. J. Orr, V. W. Sparrow, and R. C. Waag, "Simulation of ultrasonic pulse propagation through the abdominal wall," *The Journal of the Acoustical Society of America*, vol. 102, no. 2, p. 1177, 1997.

[4] C. W. Manry and S. L. Broschat, "FDTD simulations for ultrasound propagation in a 2-D breast model," *Ultrasonic Imaging*, vol. 18, no. 1, pp. 25–34, 1996.

[5] X. Yuan, D. Borup, J. Wiskin, M. Berggren, and S. Johnson, "Simulation of acoustic wave propagation in dispersive media with relaxation losses by using FDTD method with PML absorbing boundary condition," *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 46, no. 1, pp. 14–23, 1999.

[6] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," in *IEEE Int. Symp. on Parallel & Distributed Processing*. IEEE, May 2009, pp. 1–8.

[7] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[8] T. D. Mast, L. P. Souriau, D. L. Liu, M. Tabei, A. I. Nachman, and R. C. Waag, "A *k*-space method for large-scale models of wave propagation in tissue," *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 48, no. 2, pp. 341–354, 2001.

[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *ACM/IEEE Conf. on Supercomputing*. IEEE, November 2008.

[10] K. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Transactions on Antennas and Propagation*, vol. 14, no. 3, pp. 302–307, 1966.

[11] B. Fornberg, "Generation of Finite Difference Formulas on Arbitrarily Spaced Grids," *Mathematics of Computation*, vol. 51, no. 184, pp. 699—706, 1988.

[12] A. Karamalis, W. Wein, and N. Navab, "Fast Ultrasound Image Simulation using the Westervelt Equation," *Medical Image Computing and Computer-Assisted Intervention*, vol. 6361, pp. 243–250, 2010.

[13] F. Varray, C. Cachard, A. Ramalli, P. Tortoli, and O. Basset, "Simulation of ultrasound nonlinear propagation on GPU using a generalized angular spectrum method," *EURASIP Journal on Image and Video Processing*, vol. 2011, no. 1, pp. 1–6, 2011.

[14] A. A. Haigh, B. E. Treeby, and E. C. McCreath, "Ultrasound simulation on the cell broadband engine using the Westervelt equation," in *Algorithms and Architectures for Parallel Processing*, September 2012, pp. 241–252.

[15] A. Balevic, L. Rockstroh, M. Wroblewski, and S. Simon, "Using arithmetic coding for reduction of resulting simulation data size on massively parallel GPGPUs," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2008, pp. 295–302.

[16] S. Al Junid, M. Haron, Z. Abd Majid, A. Halim, F. Osman, and H. Hashim, "Development of novel data compression technique for accelerate dna sequence alignment based on smith-waterman algorithm," in *3rd UKSim European Symp. on Computer Modeling and Simulation*. IEEE Computer Society, 2009, pp. 181–186.

[17] S. Lietsch and O. Marquardt, "A CUDA-supported approach to remote rendering," in *Advances in Visual Computing*. Springer, 2007, pp. 724–733.

[18] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *Proc. of the 20th Annu. Int. Conf. on Supercomputing*. ACM, 2006, pp. 307–316.

[19] B. E. Usevitch, "JPEG2000 extensions for bit plane coding of floating point data," in *Data Compression Conf.* IEEE, March 2003, p. 451.

[20] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[21] V. Engelson, D. Fritzson, and P. Fritzson, "Lossless compression of high-volume numerical data from simulations," in *Data Compression Conf.* IEEE, March 2000, p. 574.

[22] A. Omeltchenko, T. J. Campbell, R. K. Kalia, X. Liu, A. Nakano, and P. Vashishta, "Scalable I/O of large-scale molecular dynamics simulations: A data-compression algorithm," *Computer Physics Communications*, vol. 131, no. 1, pp. 78–85, 2000.

[23] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, 2009.

[24] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 gb/s on a GPU," in *Proc. of the 4th Workshop on General Purpose Processing on Graphics Processing Units*. ACM, March 2011.

[25] A. Trott, R. Moorhead, and J. McGinley, "Wavelets applied to lossless compression and progressive transmission of floating point data in 3-D curvilinear grids," in *IEEE Symposium on Volume Visualization*. IEEE, October 1996, pp. 385–388.

[26] M. J. Gormish, E. L. Schwartz, A. F. Keith, M. P. Boliek, and A. Zandi, "Lossless and nearly lossless compression for high-quality images," in *SPIE, Very High Resolution and Quality Imaging II*. International Society for Optics and Photonics, February 1997, pp. 62–70.

[27] M. F. Hamilton and D. T. Blackstock, Eds., *Nonlinear Acoustics*. Melville: Acoustical Society of America, 2008.

[28] G. Pinton, J. Dahl, S. Rosenzweig, and G. Trahey, "A heterogeneous nonlinear attenuating full-wave model of ultrasound," *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 56, no. 3, pp. 474–488, 2009.

[29] NVIDIA, "Advanced CUDA Webinar: Memory optimizations," http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf, 2011, accessed: 2014-01-14.