Hierarchical Parallelism in a Physical Modelling Synthesis Code

Hierarchical Parallelism in a Physical Modelling Synthesis Code

James PERRY, Stefan BILBAO and Alberto TORIN
The University of Edinburgh

Abstract

Modern computer hardware provides parallelism at various different levels - most obviously, multiple multicore processors allow many independent threads to execute at once. At a finer-grained level, each core contains a vector unit allowing multiple integer or floating point calculations to be performed with a single instruction. Additionally, GPU hardware is highly parallel and performs best when processing large numbers of independent threads. At the same time, tools such as CUDA have become steadily more abundant and mature, allowing more of this parallelism to be exploited.

In this paper we describe the process of optimising a physical modelling sound synthesis code, the Multiplate 3D code, which models the acoustic response of a number of metal plates embedded within a box of air. This code presented a number of challenges and no single optimisation technique was applicable to all of these. However, by exploiting parallelism at several different levels (multithreading, GPU acceleration, and vectorisation), as well as applying other optimisations, it was possible to speed up the simulation very significantly.

1. Background

1.1. Digital Sound Synthesis

Digital sound synthesis has a long history, dating back at least as far as work at Bell Labs in the 1950s; the earliest techniques employed simple computational structures, such as circular reading from waveforms stored in buffers (wavetable synthesis), or the summation of sinusoidal tones [1]. Later refinements to such sound-producing algorithms, such as frequency modulation synthesis (FM) [2] and waveshaping methods [3] have led to the widespread use of such synthesis techniques. The main advantage of such methods is that, computationally speaking, they are extremely cheap. On the other hand, they are also not well-suited to producing sounds of a natural acoustic character; one way of addressing the synthetic character of sounds produced in this way is through the use of recorded audio fragments, or sampling, however this is very inflexible. Another is

through simulation-based approaches, whereby the equations of motion describing a vibrating system are simulated in their entirety. Such methods are referred to as physical modeling synthesis.

Many approaches to physical modeling synthesis have emerged - perhaps the best known are digital waveguides, based on the use of efficient delay line structures to simulate wave motion in 1D objects [4]; and modal methods, whereby the dynamics of a vibrating object are decomposed into modal contributions [5]. More recently, approaches based on direct time-stepping methods such as the finite difference time domain methods (FDTD) [6] have been employed - while more computationally intensive, such methods allow for the simulation of more complex instrument configurations, involving the coupling of multiple disparate components, which are often nonlinear, and also the immersion of the instrument in the 3D acoustic field [7]. NESS (Next Generation Sound Synthesis) is a project currently under way at the University of Edinburgh which is devoted to the exploration of such synthesis methods on a large scale, and for a great variety of instrument types: percussion, string, brass, electromechanical, and also 3D room simulations. As computational requirements are heavy, implementation in parallel hardware is necessary, and this forms the basis of the NESS project.

1.2. Physical model

The Multiplate 3D code [8] is a physical modelling synthesis code that simulates multiple rectangular plates, aligned horizontally within a finite box of air. The plate simulations are performed using a non-linear model based on the von Kármán equations for thin plate vibration [9]. The displacement $w(x, y, t)$ of one plate at position (x, y) and time $t \in \mathbb{R}^+$ is governed by the following partial differential equations:

$$\ddot{w} = -\kappa_1 \Delta^2 w + \mathcal{N}(w, F) + f_+ + f_- \tag{1a}$$

$$\Delta^2 F = -\kappa_2 \mathcal{N}(w, w), \tag{1b}$$

where F is an auxiliary variable called Airy's stress function, Δ^2 represents the biharmonic operator and the double dot notation indicates the second derivative with respect to time. The operator $\mathcal{N}$ is a bilinear mapping that for any two functions f, g is defined as

$$\mathcal{N}(f, g) = \partial_{xx} f \partial_{yy} g + \partial_{yy} f \partial_{xx} g - 2\partial_{xy} f \partial_{xy} g, \tag{2}$$

which introduces the nonlinearity in the model. The quantities κ_1 and κ_2 are constants defined by the physical parameters of the model, while f_+ and f_- represent the pressure exerted on the plate by the acoustic field from above and from below. Loss terms can be added to the first of (1), as well.

The acoustic field surrounding the plates can be described by the 3D wave equation [10]. Suitable boundary and coupling conditions must be imposed at the walls of the airbox and at the interfaces with the plates, respectively. More details can be found in [8].

1.3. Numerical implementation

The behaviour of the physical model described above can be calculated by discretising the underlying physical equations with the finite difference method [6]. Despite the considerable mathematical complexity of the model, the numerical implementation has a very simple structure, see Section 1.4.

When viewed as a sound synthesis tool, the code is flexible enough to accomodate the wishes of a composer. The dimensions of the airbox, as well as the sizes and positions of the plates can be specified by the user, together with all the physical parameters characterising the model. The plates and the airbox have separate co-ordinate systems, allowing the optimal grid spacing to be used for each element of the simulation, interpolations being performed between the various grids when necessary. Input to the model is provided in the form of simulated strikes on the plates, and audio outputs are taken both from points on the plate surfaces and points within the airbox. The nonlinear equations (1) produce more realistic sounds than a simple linear model, especially when higher amplitude strikes are introduced. A finite difference scheme for their solution has been presented in [11].

For the purposes of comparing the speeds of different versions of the code, we will focus on one particular test problem that is fairly typical of the problems users want to simulate: a set of three plates of different sizes (specifically, 0.81x0.87m, 0.39x0.42m, and 0.65x0.61m) embedded in an airbox measuring 1.37x1.32x1.32m. For most of the tests, the time taken to run the simulation for 50 timesteps (representing just over 1 millisecond of real time at 44.1kHz) was used to quickly compare different versions of the code.

The code was initially written in Matlab, making extensive use of sparse matrix operations for almost every operation. However, the Matlab version is extremely slow: simulating the test problem for one second takes over an hour and a half (specifically, 5877 seconds) on a typical machine, and so an optimised version is highly desirable.

For examples of sounds generated using the code, please see [12].

1.4. Code Structure

Each simulation timestep generates a single audio sample, so to run a one second simulation at CD quality (44.1kHz), 44100 timesteps must be run; rendering a four minute long composition would require over 10 million timesteps. The high number of timesteps required for practical runs makes the code quite computationally intensive despite its relatively small domain size, and limits the amount of parallelism available in the code since dependencies prevent parallelisation in the time dimension.

The main loop of the code can be divided into the following three major categories:

1. the linear system solve for each plate
2. the airbox simulation
3. the remainder

Solver	Remainder	Airbox	Total
2.40s	0.81s	2.26s	5.47s

Table 1. Profile of original Matlab implementation

The remainder mostly consists of building a system matrix for each plate that is then used in the linear system solve, but also includes other operations, such as interpolating between the plate co-ordinate system and the airbox co-ordinate system.

In the Matlab version of the code, the time spent in each of these sections for a 50 iteration run of the test problem is shown in Table 1. No single part of the code dominates the run time, so all must be optimised in order to improve the performance significantly.

The airbox simulation is an excellent candidate for GPU acceleration, being a simple linear stencil update where all points can be run in parallel. Because it is coupled to the plate simulations, it would be advantageous to also simulate the plates on the GPU to minimise the amount of data transfered across the PCI bus. However, the linear system solve operation required at each timestep for each plate - or more specifically, the preconditioner for this solver - proved extremely difficult to parallelise effectively.

2. Optimisation and Parallelisation

2.1. Solver

The original Matlab version of the code used the preconditioned conjugate gradient solver algorithm [13] with a Cholesky preconditioner. Instead of generating the preconditioner from the system matrix, which would have to be redone at every timestep, the Cholesky decomposition [14] of the biharmonic operator [15] is computed once for each plate at startup and stored for later use. This preconditioner is extremely effective for the systems in Multiplate, often allowing them to be solved in fewer than 5 iterations of the conjugate gradient solver.

For each iteration of the conjugate gradient algorithm, the preconditioner step computes two triangular solves using the Cholesky decomposition. Unfortunately the triangular solve is an inherently serial operation: every value being solved for is dependent on the value computed in the previous row, making this algorithm unsuitable for GPU acceleration or domain decomposition. While there have been attempts [16] to optimise triangular solves using CUDA, these are highly dependent on there being potential for parallelism within the matrix structure; matrices without this potential are unlikely to run well in CUDA as serial GPU performance is very poor. This was the case for our test problem, for which this method was many times slower than simply running the algorithm on a single CPU core. It was clear that this phase of the code was likely to become a bottleneck.

We initially sought an alternative preconditioner that was more easily parallelisable. However, when we tested several preconditioners provided by the

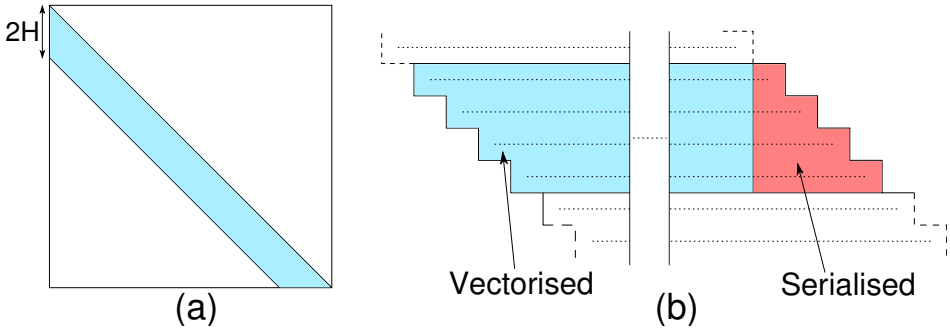

Figure 1. (a) Sparsity of Cholesky decomposition; (b) vectorising the triangular solve

PETSc library, only the ones that used triangular solves (Cholesky, LU and the incomplete variants thereof) were effective, with the other algorithms requiring hundreds of solver iterations in order to converge, and taking much more time.

We then ported the code to C and tested two further preconditioning methods: dense inverses, and sparse approximate inverses. Although for many systems a dense matrix would be too large to work with, our plates are relatively small systems so it is feasible to compute a dense inverse for the biharmonic matrix. This allows the triangular solves to be replaced with a simple dense-matrix-by-vector multiplication - a highly parallel operation that is well suited to GPU acceleration. However, due to the sheer number of operations required, it is much slower than the triangular solve, even when run on a GPU. For our test matrix, an optimised CUDA matrix-by-vector multiplication took 393μS, slower than even the unoptimised C triangular solve (220μS for both forward and back solves), although still faster than running the triangular solves in CUDA (890μS).

We also tried using a sparse approximation to the inverse of the matrix, generated by MSPAI [17], to reduce the number of operations required. Unfortunately the matrices generated this way were poor approximations and were not usable as an effective preconditioner.

There appeared to be no good alternative to the triangular solves, and so our only option was to make them run as fast as possible. It had also become clear that they were not going to run well on a GPU and would have to be kept on the CPU. Although they are unsuitable for most traditional parallelisation techniques due to dependencies between the rows, they can be sped up to some degree using SIMD (single instruction multiple data) instructions. The Cholesky decomposition of the biharmonic operator results in a banded matrix with a fully populated band extending 2H points below the diagonal (where H is the height of the plate's grid in points), but no non-zero values outside this band - see Figure 1(a). It is possible to vectorise the triangular solve, running several rows at once, with a relatively small amount of serialisation at the diagonal end - see Figure 1(b). (In practice, the ratio of vectorised time to serialised time is much greater than in the diagram, which is exaggerated for clarity).

The vast majority of Intel and AMD processors now support SSE (Streaming SIMD Extensions) and SSE2. These are 128-bit wide vector instruction sets:

Code version	Run time (μS)
Original C	110
Banded C	94
Unrolled C	72
SSE	26

Table 2. Run times of triangular solver implementations

the original SSE supported 4-way parallelism on single precision floating point numbers, with SSE2 adding 2-way parallelism for double precision numbers. The original version of the multiplate code uses double precision exclusively; although single precision would be more than adequate for the audio outputs generated by the code, using single precision for the simulation itself has been found to cause numerical instability in similar codes. However, our tests showed that single precision was adequate for the preconditioner and so 4-way SSE instructions can be used.

Before turning to SSE, the C version of the triangular solve was optimised as much as possible. The generic compressed sparse row matrix format used to store the Cholesky decomposition was replaced by a banded format, eliminating an indexing step from the inner loop. The inner loop was also unrolled to reduce loop overheads. Both of these optimisations improved the performance slightly. However the C compiler (gcc) was still not able to vectorise this routine automatically, so a hand coded implementation using SSE intrinsics was created. This version of the triangular solve is around 4.2x faster than the original C version and 2.8x faster than the optimised C version. The time taken (in microseconds) for a triangular solve of a 1155x1155 matrix (band size 70) from the test problem is shown in Table 2. The SSE triangular solve increased the speed of the entire linear system solver by 1.6x.

Newer processors also support AVX (Advanced Vector Extensions), a 256-bit wide vector instruction set supporting 4-way double precision and 8-way single precision. This has the potential to speed up some codes even more, but unfortunately it is a bad fit for the triangular solve. Our SSE implementation is quite dependent on the shuffle instructions that can permute the order of elements in a vector register, and AVX's 8-way shuffle instructions are less flexible. Although it is possible to work around the limitations, doing so negates any performance gained from using AVX and the resulting code is slower than the SSE version.

2.2. System Matrix Builder

Building the system matrix requires a complex sequence of algebraic operations which must be performed for each plate at each iteration of the timestep loop. The original Matlab version of the code uses generic sparse matrix operations to implement this, as does the subsequent unoptimised C port. However, many of the steps can be done more efficiently by taking into account the matrix structures, which stay the same even when the numbers within them change. The majority of the matrices used in the code represent some kind of stencil applied to a 2D grid

of points; square 3x3 and 5x5 stencils are the most common. These translate into diagonally banded matrices. The 3x3 stencil matrices have 3 bands (one on the diagonal and one either side), each 3 elements wide, and the 5x5 matrices have 5 bands (one on the diagonal and two either side), each 5 elements wide. There are no non-zero elements outside the bands.

Rather than storing these as generic sparse matrices, they can be stored much more efficiently by taking into account their structure: only their overall size, the distance between the bands and an array of values need to be stored, saving memory and freeing the CPU from having to work with index arrays when manipulating them. We supplemented our generic sparse matrix library with a banded matrix library, capable of storing matrices representing 3x3 and 5x5 stencils as described. The most critical and frequently used functions (banded-matrix-by-vector multiplications) were implemented as optimised SSE2 kernels. For even greater performance, some sequences of operations required for building the system matrix were rolled together into custom functions, merging loops and avoiding the need to write intermediate results back to memory. Overall, these optimisations sped up the "remainder" part of the code by 9.2x.

2.3. Airbox Update

The airbox update phase of the Multiplate code is, in isolation, an excellent candidate for GPU acceleration: it consists of the same operation performed many times across a large number of data elements, with no dependencies between them. Its integration with the plate simulations (which must run on the CPU for good performance, as described above) makes it somewhat less suitable, due to the need to transfer data across the bus to and from the GPU during the timestep loop. However, the amount of data required to be transferred proved small enough that there is still a performance advantage to running the airbox on the GPU.

A CUDA kernel was written to perform the airbox stencil update. Additionally, the interpolations between airbox and plate grids were offloaded to the GPU using the CuSparse library [18] - as these were already implemented as sparse-matrix-by-vector multiplications, this was straightforward. For the test problem, the CUDA version of the airbox update runs around 6 times faster than the single core C version, including the data transfer overheads.

It is advantageous to overlap the airbox computations on the GPU with the plate updates on the CPU and run them concurrently; this is non-trivial because the plate updates require the latest values from the airbox before they can run, but it was achieved using the following scheme. The airbox update for timestep `t+1` is run concurrently with the plate updates for timestep `t`. This means that the values from the airbox computation are always ready when the plates require them. However, airbox elements adjacent to the plates will be computed incorrectly because they depend on the plate updates for the previous timestep which have not yet completed, so these elements must be recalculated after the plate updates finish. This recalculation only affects a small fraction of the whole airbox so it does not have a significant impact on overall performance, compared with the benefit from the overlap. (Due to the handling of the domain in CUDA it is simpler and faster to simply compute these elements and then overwrite them later with the correct values, rather than omit them from the first computation).

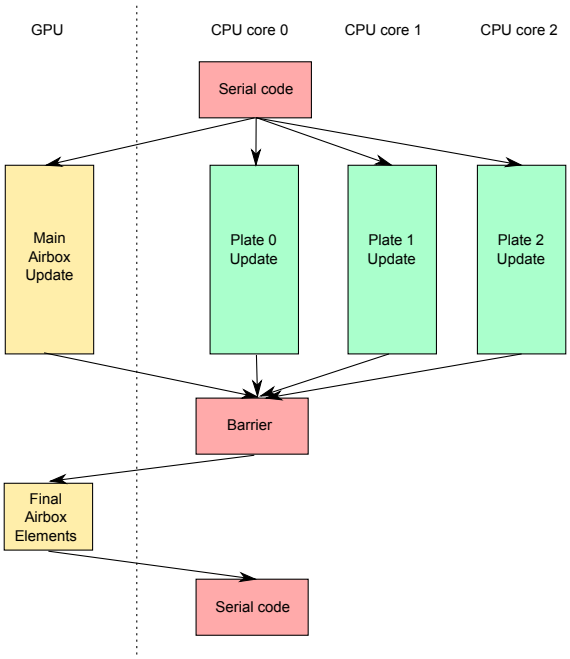

Figure 2. Execution timeline for optimised code

2.4. Parallelising the Plates

Whilst there are dependencies between each plate and the airbox, no direct dependencies exist between the individual plates. The triangular solve algorithm prevents us from running them on the GPU as described above, but they are a good candidate for running in parallel using multithreading, taking advantage of modern multicore CPUs. This was initially attempted using OpenMP, however performance was very poor due to the worker threads yielding the CPU when idle, resulting in up to 23ms being wasted every timestep on waiting for the threads to be scheduled again. There did not appear to be any portable way to force OpenMP to keep the threads running constantly, so a lower level approach using Pthreads was used instead.

The end result is that, given enough CPU cores, the plate updates all run in parallel, while most of the airbox update runs concurrently on the GPU. Typically, this results in the whole system running as fast as the largest plate simulation, with the other plates and the airbox being effectively "for free". See Figure 2.

3. Conclusion

The final optimised version of Multiplate 3D runs approximately 60-80x faster than the original Matlab version for typical problem sizes, making it much more practical for use by researchers and composers - runs which previously took hours to complete can now be done in minutes. This demonstrates that where

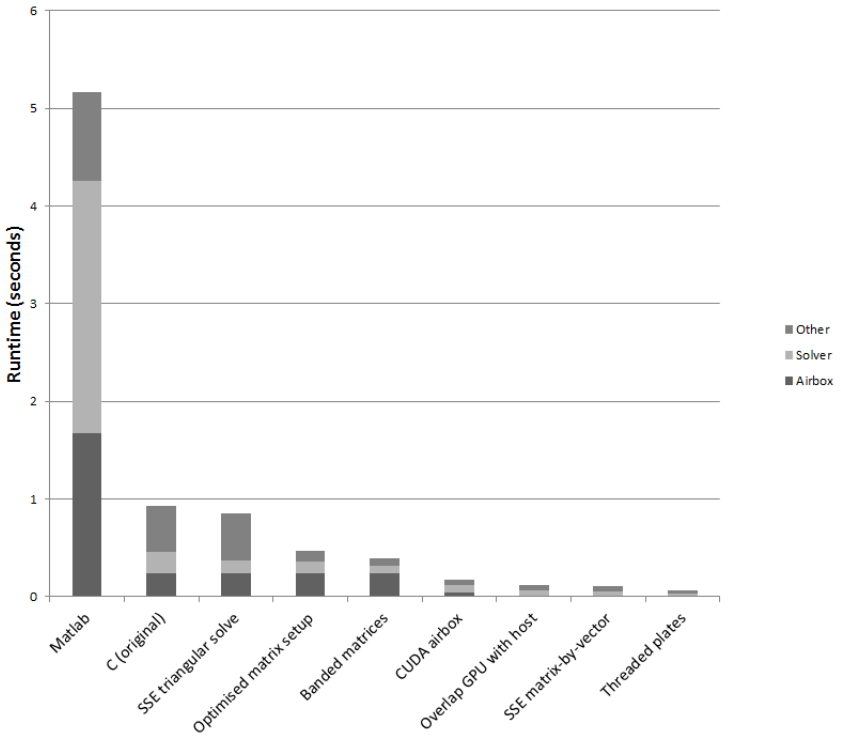

Figure 3. speeds and profiles of different versions of code

no single optimisation technique can deliver useful performance gains by itself, a combination of multiple techniques can potentially be more effective.

The overall speeds and profiles of the different versions of the code are broken down in Figure 3. It shows that the airbox simulation was sped up dramatically when switching from the sparse matrix-based version in Matlab to a simple loop in the C port, and then sped up again when porting to CUDA. Overlapping the airbox computation on the GPU with the plates on the CPU effectively hides the time spent on the airbox completely. The solver portion of the code was sped up in stages, gaining a lot of speed with the move from Matlab to custom C code, and even more when the generic C sparse matrix routines for triangular solves and matrix-vector multiplies were replaced with optimised SSE kernels operating on custom banded data structures. The remainder of the code was also markedly sped up by replacing generic sparse matrix operations with custom optimised code. Finally, running the plate simulations in parallel gave another boost to both these sections of the code.

The performance figures given in this paper relate to the `fermi1` system at EPCC. This system comprises 4 6-core Intel Xeon X5650 CPUs running at 2.67GHz, and 4 NVidia Tesla C2050 GPUs. Unless otherwise noted, CPU performance numbers are for single threaded code. GPU performance numbers are always for a single GPU as no attempt was made to take advantage of multiple

GPUs. The Matlab version of the code could not be run on the `fermi1` system, so the Matlab timings have been adjusted by multiplying them by a scaling factor to reflect the difference in performance between `fermi1` and the system they were actually run on.

4. Acknowledgements

This work was supported by the European Research Council, under grant number StG-2011-279068-NESS.

References

[1] C. Roads and J. Strawn, editors. *Foundations of Computer Music*. MIT Press, Cambridge, Massachusetts, 1985.

[2] J. Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society*, 21(7):526–534, 1973.

[3] M. LeBrun. Digital waveshaping synthesis. *Journal of the Audio Engineering Society*, 27(4):250–266, 1979.

[4] J. O. Smith III. *Physical Audio Signal Processing*. Stanford, CA, 2004. Draft version. Available online at `http://ccrma.stanford.edu/~jos/pasp04/`.

[5] D. Morrison and J.-M. Adrien. Mosaic: A framework for modal synthesis. *Computer Music Journal*, 17(1):45–56, 1993.

[6] B. Gustafsson, H.-O. Kreiss, and J. Oliger. *Time Dependent Problems and Difference Methods*. John Wiley and Sons, New York, New York, 1995.

[7] S. Bilbao. *Numerical Sound Synthesis*. John Wiley and Sons, Chichester, UK, 2009.

[8] A. Torin and S. Bilbao. A 3D multi-plate environment for sound synthesis. *Proc. of the 16th Int. Conference on Digital Audio Effects (DAFx-13), Maynooth, Ireland*, 2013.

[9] A. Nayfeh and D. Mook. *Nonlinear oscillations*. John Wiley and Sons, New York, 1979.

[10] P. M. Morse and K. U. Ingard. *Theoretical acoustics*. McGraw-Hill Inc., USA, 1986.

[11] S. Bilbao. A family of conservative finite difference schemes for the dynamical von krmn plate equations. *Numerical Methods for Partial Differential Equations*, 24(1):193–216, 2008.

[12] A. Torin. 3D multi-plate environment web page. `http://edin.ac/1TavajV`, 2013.

[13] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. `http://www.cs.cmu.edu/ ./quake-papers/painless-conjugate-gradient.pdf`, 1994.

[14] N. J. Higham. Cholesky factorization. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(2):251–254, 2009.

[15] Dr. A. P. S. Selvadurai. *Partial Differential Equations in Mechanics 2*. Springer, 2000.

[16] M. Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011-001*, 2011.

[17] A. Kallischko. *Modified Sparse Approximate Inverses (MSPAI) for Parallel Preconditioning*. Dissertation, Fakultät für Mathematik, Technische Universität München, mar 2008.

[18] NVIDIA. Cusparse library. *NVIDIA Corporation, Santa Clara, California*, 2011.