LARGE STENCIL OPERATIONS FOR GPU-BASED 3-D ACOUSTICS SIMULATIONS

Brian Hamilton[†], [*] *Craig J. Webb*[†], *Alan Gray*[‡], *Stefan Bilbao*[†]

[†]Acoustics and Audio Group,
[‡]EPCC,
University of Edinburgh

ABSTRACT

Stencil operations are often a key component when performing acoustics simulations, for which the specific choice of implementation can have a significant effect on both accuracy and computational performance. This paper presents a detailed investigation of computational performance for GPU-based stencil operations in two-step finite difference schemes, using stencils of varying shape and size (ranging from seven to more than 450 points in size). Using an Nvidia K20 GPU, it is found that as the stencil size increases, compute times increase less than that naively expected by considering only the number of computational operations involved, because performance is instead determined by data transfer times throughout the GPU memory architecture. With regards to the effects of stencil shape, performance obtained with stencils that are compact in space is mainly due to efficient use of the read-only data (texture) cache on the K20, and performance obtained with standard high-order stencils is due to increased memory bandwidth usage, compensating for lower cache hit rates. Also in this study, a brief comparison is made with performance results from a related, recent study that used a shared memory approach on a GTX 670 GPU device. It is found that by making efficient use of a GTX 660Ti GPU—whose computational performance is generally lower than that of a GTX 670—similar or better performance to those results can be achieved without the use of shared memory.

1. INTRODUCTION

At the heart of many grid-based physics simulations is a stencil operation approximating some differential operator of interest. In the context of acoustics modelling based on the 3-D wave-equation [1], the finite difference (FD) method with explicit time integration (also known as finite difference time domain (FDTD)) is a standard approach [2–4]. The stencil operation involved is typically regular over a Cartesian grid and is easily parallelised, making the FD method a suitable candidate for acceleration on graphics processing units (GPUs). General programming on high-performance computing devices has been studied by many in the context of 3-D acoustic wave equations [5–7], and more specifically for GPU-based 3-D room acoustics simulations [8–17].

It is well-known that the simplest FDTD scheme suffers from significant numerical dispersion errors. Mitigating such errors to tolerable levels may require grid refinements that can dramatically increase the computational resources required. As such, there has been much work in designing more accurate FDTD schemes, often by taking into account more than the standard number of points (seven in space and three in time); examples include explicit and implicit schemes using 27-point stencils [15, 18–20], schemes that make use of standard high-order spatial differencing [3, 21], and high-order schemes based on modified equation methods [3, 21].

Of particular interest to this study is a general construction for 3-D discrete Laplacians on the Cartesian grid recently presented [22] that allows for a general family of two-step FDTD schemes utilising stencils of almost arbitrary size and shape. This general construction is beneficial in that it provides a large set of free parameters to be optimised for the purposes of minimising dispersion or isotropy errors [22]. For example, an eight-parameter 125-point ($5\times5\times5$) stencil scheme was shown to have at most two-percent absolute dispersion error over at least 85% of the normalised wavenumbers simulated [22, Fig. 8], which is a significant improvement over conventional 27-point schemes that meet the same criterion over at most 40% of the normalised wavenumbers simulated [20]. As such, by taking on *linear* increases in operations over standard 7-point or 27-point schemes, one can meet any desired accuracy (in terms of numerical dispersion) for a given application without the usual *cubic* or *quartic* increases in computational costs associated with grid refinement, thereby approaching the minimum grid requirements dictated only by sampling considerations.

For the purposes of large-scale 3-D acoustics simulations, such large-stencil schemes should also benefit from parallel implementations and memory caches on GPU devices. It is thus of interest, and the purpose of this study, to determine how the size and shape of such stencils affects processing throughput on a GPU device, where performance is not necessarily determined by the number of operations, but rather by data movement. Related work, not specific to acoustics simulations, can be found in [23–26]. We note that boundary conditions become invariably more difficult as the stencil size increases, but as this study is a preliminary one on GPU performance, the boundary problem will not be addressed here. The organisation of this paper is as follows. Section 2 presents the two-step schemes, stencil operations, and compact stencils under study. Section 3 features the GPU kernels and testing procedures used. Results from K20 GPU tests are presented in Section 4, followed by a brief comparison with shared memory approaches using a GTX 660Ti GPU.

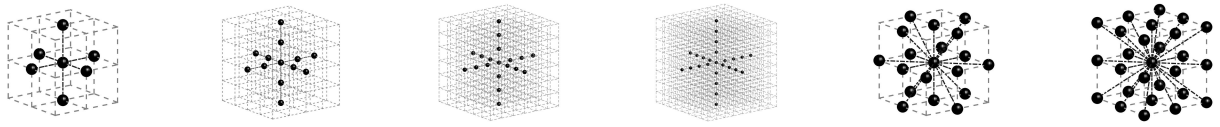
2. BACKGROUND

2.1. Two-step finite difference schemes

In order to express the two-step finite difference schemes under consideration, we define a fully-discrete Cartesian grid function

[*] This work was supported by the European Research Council, under grant StG-2011-279068-NESS, and by the Natural Sciences and Engineering Research Council of Canada. Thanks to Shiv Upadhyay for the use of the GTX 660Ti GPU card.

Email of corresponding author: brian.hamilton@ed.ac.uk.



(a) 7-point stencil (b) $M = 2$, leggy (c) $M = 3$, leggy (d) $M = 4$, leggy (e) 19-point, compact (f) 27-point, compact

Figure 1: Various stencils on the cubic lattice. These include $(6M + 1)$-point leggy stencils for $M = 1, 2, 3, 4$ and compact-in-space 19- and 27-point stencils. Bounding boxes are rescaled for clarity.

$u^n_{\boldsymbol{i}} = u(nT, \boldsymbol{i}X)$, where $n \in \mathbb{Z}^+$, T is the time-step and X is the grid spacing, and $\boldsymbol{i} = (i_x, i_y, i_z) \in \mathbb{Z}^3$ are discrete spatial indices. In this case the Cartesian grid is simply $X\mathbb{Z}^3$. An arbitrary 3-D stencil operation will be defined as:

$$\delta_{\mathbb{S},\boldsymbol{\gamma}} u^n_{\boldsymbol{i}} := \sum_{k=0}^{K-1} \gamma_k u^n_{\boldsymbol{i}+\boldsymbol{l}_k} \tag{1}$$

where $\mathbb{S} = \{\boldsymbol{l}_0, \ldots, \boldsymbol{l}_{K-1}\}$ is a 3-D stencil (a set of K points $\boldsymbol{l}_k \in \mathbb{Z}^3$) and $\boldsymbol{\gamma} = (\gamma_0, \ldots, \gamma_{K-1})$ is a vector of non-zero scalar coefficients (stencil weights). The stencil weights are generally chosen to provide an approximation to a particular spatial operator acting on u, such as, e.g., a direction derivative $\mathbf{n} \cdot \nabla u$ for $\mathbf{n} \in \mathbb{R}^3$, the 3-D Laplacian $\Delta := \partial_x^2 + \partial_y^2 + \partial_z^2$, or the biharmonic Δ^2.

This paper is concerned with two-step finite difference schemes of the form

$$u^{n+1}_{\boldsymbol{i}} = \delta_{\mathbb{S},\boldsymbol{\gamma}} u^n_{\boldsymbol{i}} - u^{n-1}_{\boldsymbol{i}} \tag{2}$$

where $\delta_{\mathbb{S},\boldsymbol{\gamma}}$ is chosen such that the above is an approximation to the 3-D wave equation with a formal accuracy that is at least second-order in time and space. Such two-step schemes require two states to be stored in memory since $u^{n+1}_{\boldsymbol{i}}$ can overwrite $u^{n-1}_{\boldsymbol{i}}$ in place. An example is the simplest scheme for the 3-D wave equation, in which $\mathbb{S}$ and $\boldsymbol{\gamma}$ are:

$$\begin{aligned} \mathbb{S} &= \{(0,0,0), (\pm 1,0,0), (0,\pm 1,0), (0,0,\pm 1)\} \\ \gamma_0 &= (2 - 6\lambda^2)\,, \quad \gamma_k = \lambda^2\,, \quad k = 1, \ldots, 6 \end{aligned} \tag{3}$$

and $\lambda := cT/X$ is the Courant number, which is generally bounded for stability as $\lambda \leq \sqrt{1/3}$, and c represents the wave speed. This scheme employs a seven-point stencil, as illustrated in Fig. 1(a). In this simple scheme, one has that

$$\frac{1}{T^2}(u^{n+1}_{\boldsymbol{i}} - \delta_{\mathbb{S},\boldsymbol{\gamma}} u^n_{\boldsymbol{i}} + u^{n-1}_{\boldsymbol{i}}) = \partial_t^2 u - c^2 \Delta u + O(T^2) + O(X^2) \tag{4}$$

where $t \in \mathbb{R}^+$ is time. Thus, the scheme provides an approximate solution $u(t, \mathbf{x})$ to the wave equation at times $t = nT$ and spatial positions $\mathbf{x} = \boldsymbol{i}X$.

The seven-point stencil in Fig. 1(a) also belongs to the family of $(6M + 1)$-point stencils, henceforth called "leggy stencils", of which examples are shown in Figs. 1(a)-1(c) for $M \leq 4$. The stencil weights for these leggy stencils can be chosen such that the accuracy of the approximate solution is of order two in time and $2M$ in space, i.e.,$(2, 2M)$-accurate. To this end, it is worth expressing these leggy schemes in a more familiar form:

$$u^{n+1}_{\boldsymbol{i}} = 2u^n_{\boldsymbol{i}} - u^{n-1}_{\boldsymbol{i}} + \lambda^2 \textstyle\sum_{d=1}^{3} \sum_{m=-M}^{M} \beta_{M,m}\, u^n_{\boldsymbol{i}+m\hat{\mathbf{e}}_d} \tag{5}$$

where $\hat{\mathbf{e}}_d$ are the standard unit vectors in $\mathbb{R}^3$. The coefficients $\beta_{M,m} = \beta_{M,-m}$ that provide $2M$th-order accuracy in space for the Laplacian in 1-D, and by extension for the 3-D Laplacian, can be found in, e.g., [3, 21, 27] along with stability conditions for the two-step schemes in [21]. Such schemes and stencils have seen use in the room acoustics literature [11] and can be regarded as special cases of the general family of schemes presented in [22]. In a more recent study they are called "large-star stencil" schemes [16]. The simplest seven-point scheme is a special case with $M = 1$ and $(\beta_{1,0}, \beta_{1,1}) = (-2, 1)$. Note that (5) is still of the form (2), since one can set $\gamma_0 = 2 - (\lambda^2/3)\beta_{M,0}$.

Other approaches to improving FDTD schemes have focussed on achieving *isotropic* error, initially with the goal of making use of frequency-warping techniques [18]. This leads to the so-called "interpolated schemes" [18–20], which employ 19- and 27-point stencils that are compact in space [28, 29]. These stencils are illustrated in Figs. 1(e) and 1(f). For this study we will consider two possible generalisations of such compact stencils, to be defined in Section 2.2.

Before proceeding, it is important to note that the purpose of this paper is not to determine optimal stencil weights, or to compare the accuracy of various compact stencils or leggy stencils. The focus here is on computational aspects of stencil operations on GPU devices. To this end, the performance metric used here will be the "compute time per node" (CTPN), where a "node" is simply a grid point. In other words, the CTPN is the time required to process a single grid point, taken as an average over many grid points and time-steps. It has been demonstrated in previous studies [13, 14] that this measure of performance, which is simply the scaled inverse of the commonly used "Megavoxels per second" metric, is more or less constant for different grid sizes within a fixed scheme, provided that a significant proportion of GPU memory is used (i.e., as long as the occupancy rate is sufficiently high). From such performance data, it is straightforward to predict final compute times (neglecting specialised boundary updates) once stencil weights and grid densities are chosen appropriately for a model problem.

2.2. 3-D compact stencils

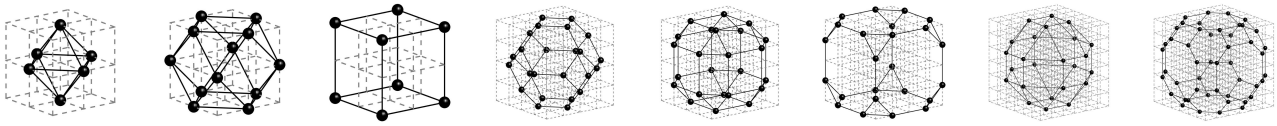
In order to construct 3-D compact stencils—beyond the standard 27-point variants [18–20, 28]—we start with a description of shells of points on the cubic lattice, $\mathbb{Z}^3$. Consider an integer-valued triplet $\boldsymbol{q} = (q_1, q_2, q_3) \in \mathbb{Q}$ with the set $\mathbb{Q}$ defined as

$$\mathbb{Q} := \{\boldsymbol{q} \in \mathbb{Z}^3 : q_1 \geq q_2 \geq q_3 \geq 0, q_1 \geq 1\} \tag{6}$$

and let $\mathcal{P}(\boldsymbol{q})$ be a function that returns the set of *unique* permutations of $(\pm q_1, \pm q_2, \pm q_3)$. Letting $|\mathcal{P}(\boldsymbol{q})|$ denote the cardinality of the set $\mathcal{P}(\boldsymbol{q})$, it is straightforward to work out that $|\mathcal{P}(\boldsymbol{q})| \in \{6, 8, 12, 24, 48\}$. Each set of triplets $\mathcal{P}(\boldsymbol{q})$ represents a shell of points on $\mathbb{Z}^3$, as illustrated in Fig. 2.

Consider now a set of P distinct triplets, $\Omega = \{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_P\}$, representing a set of shells, and define the function $\mathcal{S}(\Omega)$ as:

$$\mathcal{S}(\Omega) := (0,0,0) \cup \left(\textstyle\bigcup_{p=1}^{P} \mathcal{P}(\boldsymbol{q}_p)\right) \tag{7}$$



(a) $\boldsymbol{q} = (1,0,0)$ (b) $\boldsymbol{q} = (1,1,0)$ (c) $\boldsymbol{q} = (1,1,1)$ (d) $\boldsymbol{q} = (2,1,0)$ (e) $\boldsymbol{q} = (2,1,1)$ (f) $\boldsymbol{q} = (2,2,1)$ (g) $\boldsymbol{q} = (3,1,1)$ (h) $\boldsymbol{q} = (3,2,1)$

Figure 2: Various shells of points, $\mathcal{P}(\boldsymbol{q})$, on the cubic lattice and associated convex hulls. Bounding boxes rescaled for clarity.

Each output set $\mathcal{S}(\Omega)$ represents a 3-D stencil on $\mathbb{Z}^3$. For example, the $(6M+1)$-point leggy stencils could be written as $\mathbb{S}^{(\mathrm{l})}_M := \mathcal{S}(\Omega^{(\mathrm{l})}_M)$, where

$$\Omega^{(\mathrm{l})}_M := \{(1,0,0), \ldots, (M,0,0)\} \tag{8}$$

There are obviously many combinations of triplets $\boldsymbol{q}$ that may be used to build 3-D stencils. For this study, we consider a small subset of the many possibilities, namely two possible generalisations of the aforementioned 27-point compact stencils. The first generalisation will be called the "compact-in-space stencils", which are defined here as $\mathbb{S}^{(\mathrm{c})}_R := \mathcal{S}(\Omega^{(\mathrm{c})}_R)$, where $\Omega^{(\mathrm{c})}_R$ is a set of triplets $\boldsymbol{q}$ defined as:

$$\Omega^{(\mathrm{c})}_R := \{\boldsymbol{q} \in \mathbb{Q} : \|\boldsymbol{q}\|^2 \leq R\} \tag{9}$$

Here, R is a positive integer and $\|\boldsymbol{q}\|$ denotes the Euclidean norm of $\boldsymbol{q}$. Each set $\Omega^{(\mathrm{c})}_R$ leads to a compact-in-space stencil $\mathbb{S}^{(\mathrm{c})}_R$ whose convex hull has a circumradius of $\sqrt{R}$. Some of these compact-in-space stencils are displayed in Fig. 3, along with $K = |\mathbb{S}^{(\mathrm{c})}_R|$, the number of points in each stencil. The sequence of R values that permit valid compact-in-space stencils can be found in [30, Table 4.3]. For example, the 7-, 19-, and 27-point stencils are, respectively, $\mathbb{S}^{(\mathrm{c})}_1$, $\mathbb{S}^{(\mathrm{c})}_2$, and $\mathbb{S}^{(\mathrm{c})}_3$. It is well-known that such stencils (for $K \geq 19$) can provide isotropic error in approximations to the Laplacian [28, 29]. High orders of isotropy become easier to achieve with larger stencils and more degrees of freedom [22], and subsequently can lead to two-step schemes with high orders of accuracy (in both time and space) through the use of modified equation methods [21].

As an aside, in regards to a recent study [16] it is important to point out that, strictly speaking, the two-step leggy-stencil ("large-star") schemes ($(2,2M)$-accurate) investigated in [16] are *not* higher-order accurate for the wave equation (for all $M \geq 1$); in such schemes, the global accuracy remains second-order since the temporal error (second-order) dominates in the limit of small h provided that λ is fixed (such as, e.g., to the stability limit [16]). This fact does not appear to be taken into account in [16] when making practical comparisons to two-step schemes that are indeed higher-order accurate ($(2M,2M)$-accurate) derived using modified equation methods [15, 31]. We also note that, contrary to what is suggested in [16], the accuracy of a stencil, or more importantly, of the scheme in which it is used, is not related to its size in points in a simple manner under the modified equation framework; see [31].

Returning to the compact stencils, we note that by the nature of the Euclidean norm, the compact-in-space stencils tend towards a spherical distribution in space, as seen in Fig. 3. One could also consider stencils that are "compact" with respect to a different norm. For example, a stencil made up of a 125-point cube ($5\times5\times5$) is not compact in the sense defined above, but can be seen as "compact" in the sense that it compactly fills a cubic volume of space (i.e., compact in a Chebyshev-type norm). As such, this family of stencils, which also comprises the aforementioned 7-, 19-, and 27-point stencils, will be referred to as "box-compact". These stencils can be defined as $\mathbb{S}^{(\mathrm{b})}_{\boldsymbol{q}} := \mathcal{S}(\Omega^{(\mathrm{b})}_{\boldsymbol{q}})$ where $\Omega^{(\mathrm{b})}_{\boldsymbol{q}}$ is defined as:

$$\Omega^{(\mathrm{b})}_{\boldsymbol{q}} := \{\boldsymbol{q}' \in \mathbb{Q} : (q'_1, q'_2, q'_3) \preceq (q_1, q_2, q_3)\} \tag{10}$$

Here, $(q'_1, q'_2, q'_3) \preceq (q_1, q_2, q_3)$ means that (q'_1, q'_2, q'_3) precedes or is the same as (q_1, q_2, q_3) in a lexicographic ordering of the set $\mathbb{Q}$; i.e., $(q'_1, q'_2, q'_3) \preceq (q_1, q_2, q_3)$ iff (a) $q'_1 < q_1$ or (b) $q'_1 = q_1$ and $q'_2 < q_2$ or (c) $q'_1 = q_1$ and $q'_2 = q_2$ and $q'_3 < q_3$.

The set of compact-in-space stencils and the above-defined set of box-compact stencils have some elements in common; e.g., $\mathbb{S}^{(\mathrm{c})}_R$ with $R \in \{1,2,3,4,5,6,8,12,13,14\}$ have equivalent counterparts $\mathbb{S}^{(\mathrm{b})}_{(q_1,q_2,q_3)}$, but otherwise they lead to different stencil shapes. Some box-compact stencils are displayed in Fig. 4. Under this notation, the aforementioned 125-point stencil would be $\mathbb{S}^{(\mathrm{c})}_{(2,2,2)}$, as seen in Fig. 4(a).

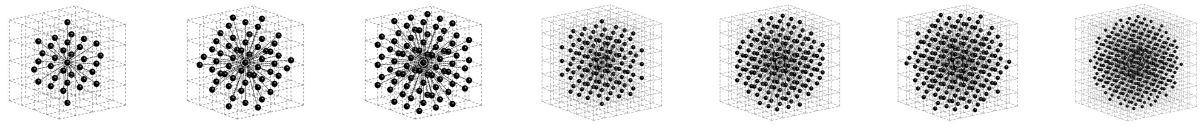
As mentioned previously, a general construction for discrete Laplacians utilising any 3-D such stencil $\mathcal{S}(\Omega)$ was presented in [22], leading to a general family of two-step schemes of which all of the examples mentioned in Section 2.1 are special cases. For brevity, the full derivation is left out, since for GPU performance comparisons (in terms of the number of grid points that may be processed in a certain amount of time) the coefficients (non-zero) are immaterial. We refer the reader to [22, Section VI] for the full derivation of the discrete Laplacian operators.

3. GPU IMPLEMENTATIONS

The GPU device that is used for the majority of the testing is an Nvidia Tesla K20 device (Kepler architecture), with error-correction checking (ECC) turned off. This card has 5119 MiB of device RAM (global memory) with a theoretical maximum memory bandwidth of 208 GB/s, although using the STREAM benchmark [32], we find that the peak copy rate is approximately 172 GB/s. The theoretical compute performance of this card is 3.52×10^{12} floating-point operations per second (FLOPS) (3.52 TFLOPS) and 1.17 TFLOPS in single and double precision, respectively. The GPU device is programmed to execute CUDA kernels.

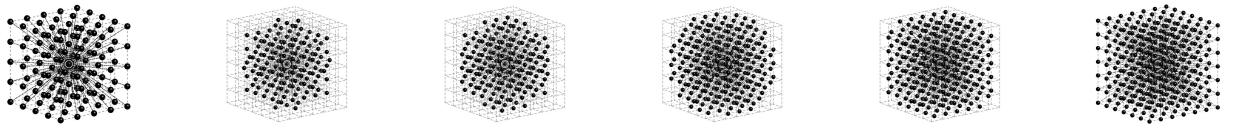
3.1. Algorithm details

Before presenting the testing procedure and the kernels under test, it is worth taking a moment to count the number of floating-point operations (FLOPs) required for the stencil operation $\delta_{\mathbb{S},\gamma} u^n_{\boldsymbol{i}}$ in (1). At first glance it appears that (1) requires K multiplications and $K-1$ additions ($2K-1$ FLOPs) per grid point. However, when fused multiply-add (FMA) instructions are available, as is often the case with GPU devices, the operation can be implemented using $K-1$ FMAs and one multiplication. Since a FMA counts as two FLOPs and only one FLOP instruction, this gives $2K-1$ FLOPs and K floating-point (FP) instructions (executed in K clock cycles) per grid point.



(a) $\mathbb{S}^{(c)}_4$, $K = 33$ (b) $\mathbb{S}^{(c)}_5$, $K = 57$ (c) $\mathbb{S}^{(c)}_6$, $K = 81$ (d) $\mathbb{S}^{(c)}_9$, $K = 123$ (e) $\mathbb{S}^{(c)}_{11}$, $K = 171$ (f) $\mathbb{S}^{(c)}_{13}$, $K = 203$ (g) $\mathbb{S}^{(c)}_{18}$, $K = 341$

Figure 3: Some compact-in-space K-point stencils $\mathbb{S}^{(c)}_R$ on the cubic lattice, reaching out a maximum squared distance R from the origin.



(a) $\mathbb{S}^{(b)}_{(2,2,2)}$, $K = 125$ (b) $\mathbb{S}^{(b)}_{(3,1,0)}$, $K = 155$ (c) $\mathbb{S}^{(b)}_{(3,1,1)}$, $K = 179$ (d) $\mathbb{S}^{(b)}_{(3,2,1)}$, $K = 251$ (e) $\mathbb{S}^{(b)}_{(3,2,2)}$, $K = 275$ (f) $\mathbb{S}^{(b)}_{(3,3,3)}$, $K = 343$

Figure 4: Some box-compact K-point stencils $\mathbb{S}^{(b)}_{\boldsymbol{q}}$ on the cubic lattice, defined by (10).

It is worth pointing out that for the discrete Laplacian operators, there are usually only $P + 1$ distinct stencil weights, where $P = |\Omega|$. In this case, the stencil operation could also be factorised and written in the form:

$$\delta_{\mathbb{S},\boldsymbol{\gamma}} u^n_{\boldsymbol{i}} = \gamma_0 u^n_{\boldsymbol{i}} + \sum_{p=1}^{P} \gamma_p \left(\sum_{\boldsymbol{l} \in \mathcal{P}(\boldsymbol{q}_p)} u^n_{\boldsymbol{i}+\boldsymbol{l}} \right) \qquad (11)$$

When carried out as suggested by the above formulation, this calculation amounts to $K - P - 1$ additions, P FMAs, and one multiply ($K + P$ FLOPs, K FP instructions) per grid point. These stencil operations, (1) and (11), would perform identically on a GPU if the performance was solely determined by the number of instructions required. However, the time taken for data to travel throughout the GPU memory architecture must also be considered. It is worth noting that the former approach leads to a simpler generalised kernel implementation (requiring only one for-loop) and it inherently utilises more of the theoretical peak FLOPS rate from the GPU. In both cases, the extra subtraction operation required for (2) adds a single FP instruction. We also note that the FLOP counts presented here differ from those recently presented [16, 17], since FMA instructions, which the tested GPUs would have invariably used, were not taken into account.

In terms of memory reads and writes, the two-step update (2) requires $K + 1$ read operations and one write operation (not counting reading $\boldsymbol{l}_k$ and $\boldsymbol{\gamma}$). As mentioned previously, the storage requirements of the scheme are $2N$ for a grid of size N, since u^{n-1} can be overwritten by u^{n+1} after being read. The K read operations required for $\delta_{\mathbb{S},\gamma} u^n_{\boldsymbol{i}}$ are not followed by overwrites, which allows for the use of read-only data cache optimisations, as will be explained shortly. There is also a requirement to store the K coefficients in $\boldsymbol{\gamma}$, or only the distinct $P+1$ if so desired, but this storage is negligible because, generally, $N \ggg K$.

The leggy-stencil family of schemes is also tested alongside the compact stencils in order to determine if the varying shape of the stencil plays a significant role in GPU performance. While these leggy schemes are encapsulated by (2) and (1), a more specialised kernel implementation is suggested by (5) that could lead to improvements in memory coalesced reads on the GPU. However, this comes at the cost of a slight redundancy, since $u^n_{\boldsymbol{i}}$ will be read and operated on three times instead of one. This alternative calculation for the stencil operation requires $K + 2$ FMAs, and one extra addition is required for the two-step update, for a total of $K + 3$ FP instructions ($2K + 5$ FLOPs) per grid point.

3.2. Testing procedure

The first twenty stencils in each family of stencils ($\mathbb{S}^{(c)}_R$, $\mathbb{S}^{(b)}_{\boldsymbol{q}}$, and $\mathbb{S}^{(l)}_M$) are tested for a box-shaped grid; the largest of these stencils is $\mathbb{S}^{(c)}_{22}$ with $K = 461$. Since the K20 card is capable of performing single and double precision FLOPs, two sets of grid sizes are used in order to use up all or half of the available memory. When testing single precision floating point, the dimensions of the grid ($N_x \times N_y \times N_z$) is either 928×800×750 or 720×640×560 and when double precision is being tested, the dimensions of the grid is 672×660×600 or 640×480×420. For each precision level, the former grid size is called "medium" and the latter "large". In each case, the grid is supplemented by a halo of N_h ghost-points (the minimum number required, i.e., the inradius of the bounding box for a given stencil) that are not updated, and thus do not factor into CTPN averages. The command-line profiler `nvprof` is used to obtain accurate estimations of average kernel execution times through the use of its kernel replay mode.

3.3. GPU Kernels

The first GPU kernel under test, `KernelA`, is a straightforward and general implementation of the two-step update (2). It appears in Fig. 5. In this kernel implementation, the grid is of size `Nxh`×`Nyh`×`Nzh`, which includes the `Nh`-thick halo of ghost points. These integer values are defined using C-preprocessor pragma statements (defined with # operator), along with `K`, the number of points in the stencil, and `RealL`, which can be defined (with #) as either `float` or `double`. The arrays `gamma` (of type `RealL`) and `offset` (of type `int`) represent stencil weights and linear offsets (linear decompositions of $\boldsymbol{l}_k \in \mathbb{S}$), respectively. These arrays are stored in the 64 KB of read-only constant memory available on the chip and accessible to all threads. Since the stencil weights and initial grid values are not important for performance analysis, the two states (`u0` and `u1`) and the stencil weights are assigned random floating-point values in the setup CPU host-code, which is also where the array `offset` is loaded with the appropriate linear offsets for the stencil under test.

A maximum threading (3-D tiling) approach is employed in this study, as implemented in lines 6-8 in Fig. 5, as opposed to, e.g., a 2-D slicing or 2-D tiling approach (see [33] for more details). In this case, the 3-D grid is decomposed in all three dimensions, with each 3-D subset of the grid assigned to a CUDA 3-D thread

```
1  __global__ void
       KernelA(ReaL *u0, const ReaL * __restrict__ u1){
2    //u1 is read from read-only data (texture) cache
3    //Nh,Nxh,Nyh,Nzh,K are #-defined ints
4    //ReaL gamma[K] is stored in constant memory
5    //int offset[K] is stored in constant memory
6    int ix = blockIdx.x*blockDim.x + threadIdx.x + Nh;
7    int iy = blockIdx.y*blockDim.y + threadIdx.y + Nh;
8    int iz = blockIdx.z*blockDim.z + threadIdx.z + Nh;
9    //condition to prevent illegal memory accesses
10   if (ix<(Nxh-Nh) && iy<(Nyh-Nh) && iz<(Nzh-Nh)){
11      //get linear index of current point
12      int cp = iz*Nxh*Nyh + iy*Nxh + ix;
13      //read previous value from global memory
14      ReaL tmp = -u0[cp];
15      //general stencil operation
16      for(int k=0;k<K;k++){
17         tmp += gamma[k]*u1[cp+offset[k]];
18      }
19      //write final value back to global memory
20      u0[cp] = tmp;
21   }
22 }
```

Figure 5: `KernelA`: CUDA kernel for compact and leggy stencils. See inline comments for implementation details.

```
1  __global__ void
       KernelB(ReaL *u0, const ReaL * __restrict__ u1){
2    //u1 is read from read-only data (texture) cache
3    //Nh,Nxh,Nyh,Nzh,K are #-defined ints
4    //ReaL gamma[K] is stored in constant memory
5    //int offset[K] is stored in constant memory
6    int ix = blockIdx.x*blockDim.x + threadIdx.x + Nh;
7    int iy = blockIdx.y*blockDim.y + threadIdx.y + Nh;
8    int iz = blockIdx.z*blockDim.z + threadIdx.z + Nh;
9    //condition to prevent illegal memory accesses
10   if (ix<(Nxh-Nh) && iy<(Nyh-Nh) && iz<(Nzh-Nh)){
11      //get linear index of current point
12      int cp = iz*Nxh*Nyh + iy*Nxh + ix;
13      //read previous value from global memory
14      ReaL tmp = -u0[cp];
15      //leggy stencil operation
16      for(int m=-Nh;m<=Nh;m++){
17         ReaL gamma_m=gamma[abs(m)];
18         tmp += gamma_m*u1[cp+m];
19         tmp += gamma_m*u1[cp+m*Nxh];
20         tmp += gamma_m*u1[cp+m*Nxh*Nyh];
21      }
22      //write final value back to global memory
23      u0[cp] = tmp;
24   }
25 }
```

Figure 6: `KernelB`: Alternative CUDA kernel for leggy stencils. See inline comments for implementation details.

block, with each thread operating on a single grid point. Enough thread blocks are issued to cover the 3-D grid, not including the halo of ghost points. The thread block size used for this test was 32×4×2, which generally provided high occupancy rates. The conditional statement at line 10 in the kernel ensures that illegal memory accesses are not encountered when the grid size is not an integer multiple of the thread-block size.

It is important to point out the `__const` and `restrict` qualifiers used for the variable `u1`, which represents u_i^n. These qualifiers signal to the compiler (`nvcc`) to make use of the read-only data cache (texture cache) for the array `u1`, which can be loaded directly from the L2 cache [34]. The L1 cache does not cache global memory reads in the K20 GPU, but the L1 cache can be configured as a shared memory unit. Shared memory implementations, while commonly used for leggy stencil schemes [5, 16], are not directly explored in this study due to space constraints, although more will be said about shared memory approaches in Section 4.3.

The second kernel under test, `KernelB`, appearing in Fig. 6, is specific to the $(6M+1)$-point leggy stencils and is meant to resemble the specific formulation (5). In this kernel, `Nh` is equal to M, and for consistency with the 3-D wave equation `gamma[m]` would be equal to $\beta_{M,m}$ for $m \geq 1$ and `gamma[0]` would be equal to $2/3 - \lambda^2\beta_{M,0}$.

4. RESULTS AND DISCUSSION

4.1. Timings

The timing results from the tests are displayed in Fig. 7. Starting with the compact stencils in Fig. 7(a), we note that, for the most part, the average compute times per node (CTPNs) scale linearly with the stencil size. As expected, single precision results in faster compute times than double precision, aside from the 27-point compact stencil which has similar performance in both precisions. Also as expected, the CTPNs are relatively invariant to the two grid sizes (medium and large), justifying the discussion at the end of Section 2.1. Within each precision, we note that it makes little difference whether the stencil is "compact-in-space" or "box-compact", aside from dips in the CTPNs in double precision for the box-compact $M \times M \times M$ stencils, $\mathbb{S}^{(b)}_{(M,M,M)}$, which is perhaps due to more efficient memory access patterns.

The leggy stencil CTPNs are displayed in Fig. 7(b) for the two different kernels. It can be seen that for leggy stencils, `KernelB` generally provides slightly faster CTPNs than `KernelA`. We also note that there are dips in the CTPNs in double precision when M is a multiple of eight, which may be due to increased memory coalescing.

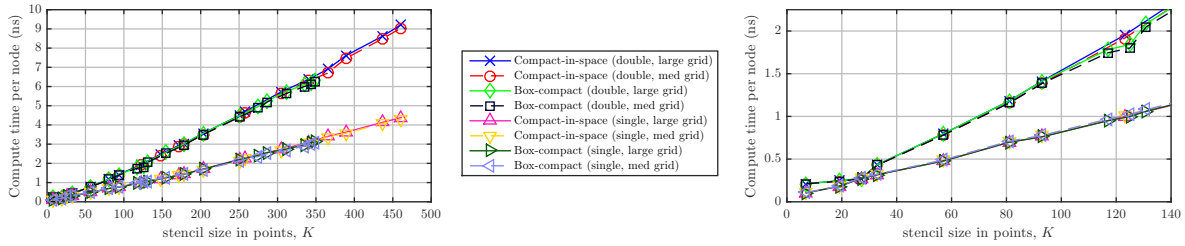
Fig. 7(c) presents a comparison of the compact stencils and the leggy stencils (with `KernelB`) in single and double precision, along with a prediction of CTPNs, stemming from increases over the simplest 7-point scheme in memory reads or in FP instructions. All of the obtained CTPNs are lower than what would be expected from this somewhat naive prediction. Finally, we note that the CTPNs vary slightly between leggy and compact stencils, but overall, the CTPNs are relatively invariant to the shape of the stencil.

In order to illustrate the utility of these timing results, it is worth going back to the example comparison in Section 1 between a 27-point scheme and a 125-point scheme. If a two-percent dispersion error were desired across some fixed bandwidth of interest, it can be calculated that the 125-point scheme with the optimised parameters used in [22] would require approximately 20× fewer pointwise updates than a 27-point scheme.[1] This outweighs the increases in CTPNs that would be seen on the K20 GPU going from a 27-point stencil to a 125-point stencil, which are 3.6× and 6.6×, respectively for single and double precision. One can carry out this type of calculation for other stencil sizes and associated schemes after analysing their dispersion errors, following [22].

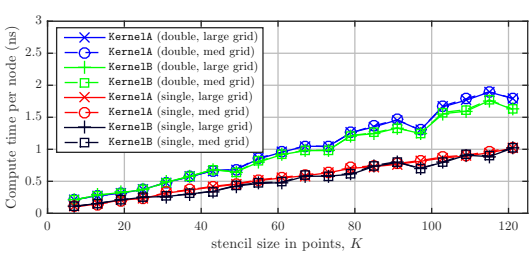
4.2. Memory throughput

In order to explain some of the timings observed, it is worth looking deeper into the many kernel metrics that can be obtained from Nvidia's CUDA profiler, and in particular, a subset relating to

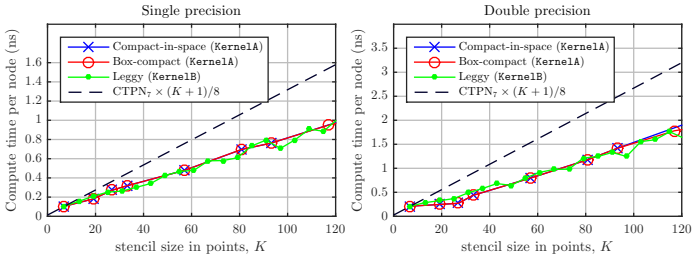
[1] The 27-point scheme would need to be oversampled by $85/40 \approx 2.1$ times that of the 125-point scheme and $2.1^4 \approx 20$.



(a) Compact stencils, double and single precision, large and medium grid sizes, using `KernelA`. Right: Plot on left for $K < 140$ in order to show detail.



(b) Leggy stencils, double and single precision, large and medium grid sizes, and the two kernel implementations.



(c) Compact and leggy stencils in single (left) and double (right) precision.

Figure 7: Average compute times per node for compact-in-space, box-compact, and leggy stencils as a function of stencil size K, on Nvidia Tesla K20 GPU card in double and single precision, and for large and medium grid sizes. In Fig. 7(c) is the line CTPN$_7 \times (K+1)/8$, where "CTPN$_7$" is the compute time per node for the 7-point stencil in either double or single precision. This line is what would be expected if compute-time increases were equal to the increases in FP instructions or memory reads required.

memory throughput. As an aside, it is worth mentioning that the following analysis will be more informative and rigorous than an approach to performance analysis recently proposed in [17, Section V], which does not make direct use of profiler metrics.[2]

Before continuing, it is worth explaining the ideal memory-reading scenario (without utilising shared memory) within this Kepler GPU memory architecture. Each stencil operation requires $K+1$ memory reads, but neighbouring points share many of these values. In the ideal scenario, each thread would read `u[cp]` from global memory (in coalesced reads) via the L2->L1 pipeline and `uz[cp]` from global memory via the L2->texture pipeline, and the remaining $K-1$ grid points would be read directly from texture cache. Once the calculation is finished, the final value would be written via L1->L2 back to global memory to be stored at `u[cp]`. So in the ideal case, the global memory read throughput would be equal to two times the global memory write throughput. However, the texture cache is not unlimited in size (48 KB per SMX), so in practice, grid values stored in texture cache that have not been completely utilised may be overwritten and reloaded into the texture cache, fetched either directly from L2 cache or from global memory, at a later stage in the stream. Thus, cache hit ratios will be lower than the ideal 100%.

With that said, Fig. 8 displays the texture cache (read-only) throughput, the L2 cache read throughput, the global memory read and write throughputs, and the global memory bandwidth used (global read + global write), for the compact-in-space (using `KernelA`) and leggy stencils (using `KernelB`) on the large grid sizes. It can be seen in Fig. 8 that the compact and leggy stencils use up nearly all of the available texture cache (read-only data cache) bandwidth (approximately 1.1 TB/s) in single precision, and a large proportion of the available texture cache bandwidth in double precision. However, the global memory read throughput is greater than two times the global memory write throughput, so the ideal scenario is not achieved. For the compact stencils, the global memory read-to-write ratio is approximately three for $K \leq 27$, and is approximately four for $K > 27$. For the leggy stencils, it is much higher, starting around three and going as high as 25, indicating a low efficiency in terms of texture cache usage. The texture throughput in double precision is about 60% that in single precision, which is a consequence of the fact that the texture cache can store more single-precision values than it can double-precision values.

The texture cache hit ratios and L2 hit ratios (from texture reads) are plotted in Fig. 9 (for the large grid sizes). It can be seen from Fig. 9 that the cache hit ratios are higher overall for the compact stencils than the leggy ones, and the L2 hit ratios (from texture reads) increase when the texture cache hit ratios decrease, indicating that many of the data not found in the texture cache were found directly in the L2 cache without having to read from global memory. On the other hand, for the leggy stencils the texture hit ratios are much lower and the L2 hit ratios (from texture reads) are also low, which means that global memory was read more often. Thus, we can conclude that the compact stencils make better use of the texture cache than the leggy stencils, and the leggy stencils are able to achieve similar performance (in terms of CTPNs) due to an increased global memory read throughput.

Aside from the 7-point scheme, the global memory bandwidth usage for compact stencils is a small fraction of the maximum theoretical bandwidth available (208 GB/s). Thus, the implemented compact stencil operations are bound by texture cache bandwidth, rather than by global memory bandwidth (aside from the 7-point scheme). On the other hand, the implemented leggy stencil operations are bound by both texture and global memory bandwidths,

[2] We note the performance metric recently proposed in [17, Section V] is based on an assumption that respective times spent on FLOPs and data transfers are additive, but this assumption does not necessarily hold for a GPU device. More importantly, the proposed metric assumes that all data transfers make use of global memory bandwidth, however, as will be seen here, data transfers in the GPU are more nuanced for such schemes—one must consider the various levels of caches that connect the global memory to SMX registers and the potential for stencil operations to make use of these caches.

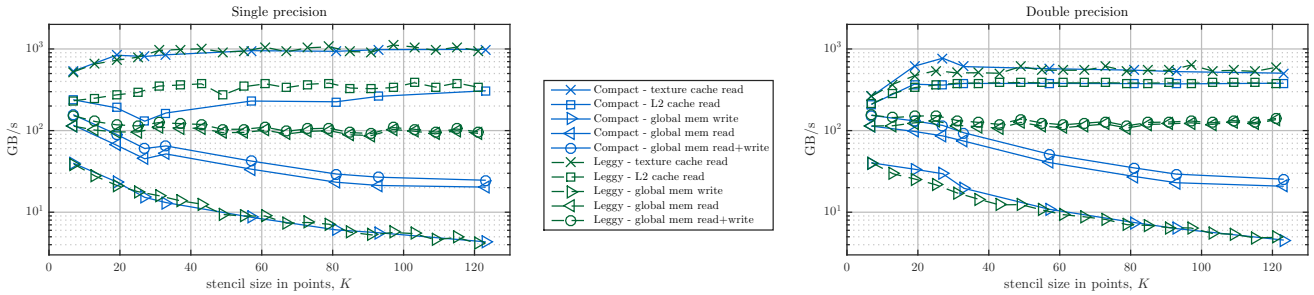


Figure 8: Memory throughput for compact-in-space stencils (using `KernelA`) and leggy stencils (using `KernelB`) as a function of stencil size in points for $K \leq 125$. Left: single precision, right: double precision.

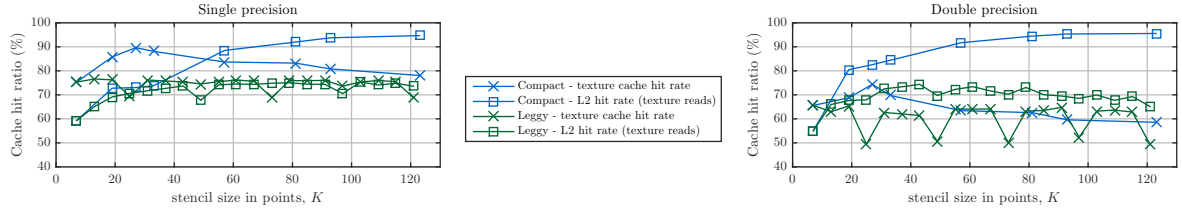


Figure 9: Texture and L2 cache hit-rates for compact-in-space stencils (using `KernelA`) and leggy stencils (using `KernelB`) as a function of stencil size in points for $K \leq 125$.

since they use a large majority of both.

Although not shown in any figures, it is worth pointing out that in all cases, occupancy rates were high (above 85%) and multiprocessor activity was nearly 100%. Also, the number of FLOPs and FLOP instructions obtained from profiler metrics agreed with the FLOP counts provided in Section 3.1. As such the FLOPS usage was at most 8% and 15% of the theoretical peak FLOPS in single and double precision respectively. Clearly then, the bottleneck is data movement rather than compute. It is also worth mentioning that the kernels were also tested without the use of `__cont restrict`, utilising the L2->L1 pipeline, and performance was significantly worse.

4.3. Comparisons with shared memory approaches

While shared memory approaches will not be directly investigated in this study, a brief comparison will be made with recently published results. In [16], leggy stencils up to $M = 8$ and a 27-point scheme were implemented on a Nvidia GTX 670 GPU device[3] following the shared memory approach of Micikevicius [5] for leggy stencils, which was also adapted to the 27-point compact stencil, although kernel codes were not provided. Texture cache was also employed, presumably to load values into shared memory.

The testing procedure in [16] was essentially the same as the one that has been used here, except that the dimensions of the box-domain were specified in metres, so grid sizes varied for each scheme with respect to specific choices of c, T, X appropriate for the schemes. A GTX 670 device was not available for a direct comparison, but a closely-related, and lower performing, GTX 660Ti GPU device was on hand. The GTX 660Ti has the same underlying chip (GK104) and the same compute specifications as the GTX 670 (clock speeds, peak FLOPS, etc.), but the GTX 660Ti is handicapped to 75% of the global memory bandwidth and L2 cache available in the GTX 670, due to one of four memory controllers on the GK104 chip being disabled.

[3] Although the GTX 670 device was not explicitly named in [16], its use was confirmed through correspondence with the authors.

Table 1: Timings for "medium-sized room" tests conducted on GTX 670 with shared memory approach in [16] alongside timings obtained here on GTX 660Ti using a maximum threading approach and pure texture fetching.

stencil	grid dims	GTX 670 [16] CTPN (ns)	GTX 660Ti CTPN (ns)	ratio
leggy, $M = 1$	$370\times259\times148$	0.239	0.162	1.47
leggy, $M = 2$	$321\times224\times128$	0.270	0.256	1.06
leggy, $M = 3$	$301\times211\times120$	0.318	0.271	1.18
leggy, $M = 4$	$290\times203\times116$	0.347	0.366	0.95
leggy, $M = 5$	$283\times198\times113$	0.424	0.375	1.13
leggy, $M = 6$	$278\times195\times111$	0.471	0.441	1.07
leggy, $M = 7$	$275\times192\times110$	0.490	0.574	0.85
leggy, $M = 8$	$272\times190\times108$	0.430	0.567	0.76
27-point compact	$642\times449\times256$	0.725	0.294	2.46

With the GTX 660Ti, the `__const restrict` qualifiers do not enable the read-only data cache loading behaviour as they do with the Tesla K20 device [34], since the compute capability of the GTX 660Ti is only version 3.0. As such, the kernels were modified to make use of texture cache bindings. The block size chosen for the GTX 660Ti was $32\times8\times4$. Using `KernelA`, we repeated the "medium-sized room" and "large room" tests with the grid dimensions listed in [16, Table 3]. The timings obtained on the GTX 660Ti for the medium-sized room are presented in Table 1, alongside the results from [16], where the CTPNs were calculated from the grid sizes and the "frames per second" values in [16, Table 3]. Ratios between timings on the two cards are also provided for comparison purposes. For brevity, the "large room" performance results are left out because they do not vary significantly from Table 1 in terms of compute times per node, both with the GTX 670 timings reported in [16] and with the GTX 660Ti.

From Table 1 it can be seen that in all cases, the performance obtained on the GTX 660Ti is at least 75% of the performance obtained on the GTX 670 in [16], and for the most part, better performance was achieved on the GTX 660Ti without the use of shared memory. Most notably, the last row in Table 1 demonstrates a significant speed-up ($2.46\times$) for the 27-point compact stencil on the GTX 660Ti in comparison to the GTX 670 timings reported in [16]. This suggests either that the shared memory implementation for

the 27-point scheme in [16] was suboptimal, or that the 27-point scheme cannot make efficient use of shared memory on these cards.

5. CONCLUSIONS AND FINAL REMARKS

In this paper, compact stencils on the 3-D cubic lattice and associated two-step finite difference schemes were analysed on GPU devices, as well as standard high-order (leggy) stencil two-step schemes. It was found that GPU performance, measured in terms of compute times per node, scaled linearly with stencil size, but generally the increases were less than what would be expected from increases in operation count over smaller stencils. It was found that data movement, rather than compute, was the bottleneck, and as such, the performance obtained can be attributed to the effects of the L2 and texture caches on the Tesla K20 card. Performance, in terms of compute times per node, varied little with respect to stencil shape for a fixed stencil size.

While overall performance did not vary with shape, the usage of memory bandwidths varied significantly between compact and leggy stencils. Compact stencils were found to make more efficient use of texture cache (higher hit rates) than leggy stencils, thus requiring fewer reads from global memory. The leggy stencil schemes required a significant portion of global memory bandwidth in order to achieve similar performance as compact stencils of similar size in points. Accordingly, it was seen that leggy stencils had relatively low L2 and texture cache hit rates.

Finally, a brief comparison was made with recently reported GPU timing results that used shared memory approaches for leggy stencil schemes ($M \leq 8$) and a 27-point compact stencil scheme. It was found that similar or better performance to the GTX 670 results reported in [16] could be obtained for most of the leggy stencil schemes considered, using a GTX 660Ti (with only 75% of the memory bandwidth and L2 cache of the GTX 670) and without the use of shared memory. For the compact 27-point scheme, a speed-up of 2.46$\times$ was achieved on the GTX 660Ti card over the reported timings from a GTX 670.

In future work, shared memory approaches could be investigated in the context of larger compact stencils, since shared memory presents many opportunities for performance. However, it is clear from these results that good performance can be obtained through pure texture fetching. It is worth recalling that the texture cache is easily accessed in newer generation Nvidia cards through the `__const restrict` qualifiers (or `__ldg()` intrinsics), which enables simpler kernel codes than those that would make use of shared memory (e.g., `KernelA` has, essentially, ten lines of code and `KernelB` has thirteen; compare to, e.g., [5, Appendix A]).

Other avenues for future work, in terms of GPU implementations, include 2-D slicing approaches, varying the order of offsets in the loop, unrolling stencil operations, and L1 caching of global memory on more recent cards than the K20. Finally, we note that maintaining good GPU performance overall with boundary condition for such schemes, which constitutes an important open problem, will present further challenges.

The C/CUDA codes used in this study are available at: `http://www2.ph.ed.ac.uk/~s1164563/dafx15`.

6. REFERENCES

[1] P. M. Morse and K. U. Ingard, *Theoretical acoustics*. Princeton University Press, 1968.

[2] R. D. Richtmyer and K. W. Morton, *Difference methods for initial-value problems*, 1st ed. Interscience Publishers, 1957.

[3] M. A. Dablain, "The application of high-order differencing to the scalar wave equation," *Geophysics*, vol. 51, no. 1, pp. 54–66, 1986.

[4] D. Botteldooren, "Acoustical finite-difference time-domain simulation in a quasi-Cartesian grid," *JASA*, vol. 95, pp. 2313–2319, 1994.

[5] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proc. of 2nd Workshop on GPGPU*. ACM, 2009, pp. 79–84.

[6] V. Etienne, T. Tonellot, P. Thierry, V. Berthoumieux, and C. Andreolli, "Optimization of the seismic modeling with the time-domain finite-difference method," in *SEG Annual Meeting*, 2014.

[7] J. Shragge, "Solving the 3D acoustic wave equation on generalized structured meshes: A finite-difference time-domain approach," *Geophysics*, vol. 79, no. 6, pp. T363–T378, 2014.

[8] N. Röber, M. Spindler, and M. Masuch, "Waveguide-based room acoustics through graphics hardware," in *Proc. Int. Computer Music Conf.*, 2006.

[9] J. Sheaffer and B. Fazenda, "FDTD/K-DWM simulation of 3D room acoustics on general purpose graphics hardware using compute unified device architecture (CUDA)," in *Proc. Institute of Acoustics*, vol. 32, no. 5, 2010.

[10] L. Savioja, "Real-time 3D finite-difference time-domain simulation of low-and mid-frequency room acoustics," in *Proc. Digital Audio Effects (DAFx)*, vol. 1, Graz, Austria, 2010, p. 75.

[11] R. Mehra, N. Raghuvanshi, L. Savioja, M. C. Lin, and D. Manocha, "An efficient GPU-based time domain solver for the acoustic wave equation," *Applied Acoustics*, vol. 73, no. 2, pp. 83–94, 2012.

[12] J. J. López, D. Carnicero, N. Ferrando, and J. Escolano, "Parallelization of the finite-difference time-domain method for room acoustics modelling based on CUDA," *Mathematical and Computer Modelling*, vol. 57, no. 7, pp. 1822–1831, 2013.

[13] J. Saarelma, "Finite-difference time-domain solver for room acoustics using graphics processing units," Master's thesis, Aalto University, 2013.

[14] B. Hamilton and C. J. Webb, "Room acoustics modelling using GPU-accelerated finite difference and finite volume methods on a face-centered cubic grid," in *Proc. Digital Audio Effects (DAFx)*, Maynooth, Ireland, Sep. 2013, pp. 336–343.

[15] B. Hamilton, S. Bilbao, and C. J. Webb, "Revisiting implicit finite difference schemes for 3-D room acoustics simulations on GPU," in *Proc. Digital Audio Effects (DAFx)*, Erlangen, Germany, 2014.

[16] J. van Mourik and D. Murphy, "Explicit higher-order FDTD schemes for 3D room acoustic simulation," *IEEE/ACM Trans. Audio, Speech, and Language Processing*, vol. 22, no. 12, pp. 2003–2011, 2014.

[17] C. Spa, A. Rey, and E. Hernandez, "A GPU implementation of an explicit compact FDTD algorithm with a digital impedance filter for room acoustics applications," *IEEE/ACM Trans. Audio, Speech, and Language Processing*, vol. 23, no. 8, pp. 1368–1380, 2015.

[18] L. Savioja and V. Valimaki, "Interpolated 3-D digital waveguide mesh with frequency warping," in *Proc. IEEE ICASSP*, vol. 5. IEEE, 2001, pp. 3345–3348.

[19] S. Bilbao, "Wave and scattering methods for the numerical integration of partial differential equations," Ph.D. thesis, Stanford University, 2001.

[20] K. Kowalczyk, "Boundary and medium modelling using compact finite difference schemes in simulations of room acoustics for audio and architectural design applications," Ph.D. dissertation, Queen's University Belfast, 2008.

[21] G. Cohen, *Higher-order numerical methods for transient wave equations*. Springer-Verlag, 2002.

[22] S. Bilbao, "Optimized FDTD schemes for 3-D acoustic wave propagation," *IEEE Trans. Audio, Speech, and Language Processing*, vol. 20, no. 5, pp. 1658–1663, 2012.

[23] D. Unat, X. Cai, and S. B. Baden, "Mint: realizing CUDA performance in 3D stencil methods with annotated C," in *Proc. Int. Conf. Supercomputing*. ACM, 2011, pp. 214–224.

[24] M. Krotkiewski and M. Dabrowski, "Efficient 3D stencil computations using CUDA," *Parallel Computing*, vol. 39, no. 10, pp. 533–548, 2013.

[25] A. Vizitiu, L. Itu, C. Nita, and C. Suciu, "Optimized three-dimensional stencil computation on Fermi and Kepler GPUs," in *IEEE HPEC*, 2014, pp. 1–6.

[26] S. Leclaire, M. El-Hachem, J.-Y. Trépanier, and M. Reggio, "High order spatial generalization of 2D and 3D isotropic discrete gradient operators with fast evaluation on GPUs," *J. Sci. Comp.*, vol. 59, no. 3, pp. 545–573, 2014.

[27] B. Fornberg, "Generation of finite difference formulas on arbitrarily spaced grids," *Mathematics of Computation*, vol. 51, no. 184, pp. 699–706, 1988.

[28] W. F. Spotz and G. F. Carey, "A high-order compact formulation for the 3D Poisson equation," *Num. Methods for PDEs*, vol. 12, no. 2, pp. 235–243, 1996.

[29] M. Patra and M. Karttunen, "Stencils with isotropic discretization error for differential operators," *Num. Methods for PDEs*, vol. 22, no. 4, pp. 936–953, 2006.

[30] J. Conway and N. J. A. Sloane, *Sphere packings, lattices and groups*, 3rd ed. Springer-Verlag, 1991.

[31] S. Bilbao and B. Hamilton, "Construction and optimization techniques for high order schemes for the two-dimensional wave equation," in *Proc. Int. Cong. Acoustics (ICA)*, Montréal, Canada, 2013.

[32] J. D. McCalpin, "STREAM: Sustainable memory bandwidth in high performance computers," University of Virginia, Tech. Rep., 2007. [Online]. Available: http://www.cs.virginia.edu/stream/

[33] C. J. Webb, "Parallel computation techniques for virtual acoustics and physical modelling synthesis," Ph.D. thesis, University of Edinburgh, 2014.

[34] NVIDIA Corporation, *CUDA C Programming Guide*. PG-02829-001_v7.0, Mar. 2015.