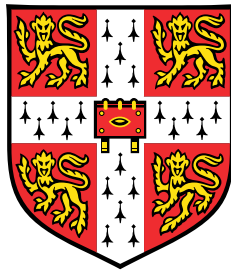


Fast finite element methods for focused ultrasound applications



Mohd Adeeb Bin Mohd Arif Kor

Supervisor: Prof. G. N. Wells

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the preface and specified in the text. It is not substantially the same as any work that has already been submitted, or is being concurrently submitted, for any degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

Mohd Adeeb Bin Mohd Arif Kor
August 2024

Abstract

This thesis presents the development of a three-dimensional, time-domain, fast finite element solver for focused ultrasound (FUS) applications.

FUS is a rapidly evolving medical technology used for tissue heating, ablation, and neuromodulation. In recent years, interest in FUS has grown significantly, especially for the treatment of cancers and brain diseases. Effective patient-specific treatment planning can be supported by accurate simulations. However, this poses a challenge from a computational standpoint since the ratio of the domain size to the wavelength is typically large, therefore computationally demanding. This is made even more challenging for nonlinear models such as the Westervelt model, commonly used in high-intensity focused ultrasound (HIFU) treatments, as higher harmonics need to be resolved.

To address this challenge, a three-dimensional, time-domain acoustic solver for FUS applications is developed. The solver is based on the high-order finite element method and the explicit Runge–Kutta method. It is well known that low-order finite element methods suffer from pollution errors, especially for high-frequency problems. However, it is demonstrated that with high-order finite elements, the pollution error is significantly reduced. Moreover, the availability of modern open-source finite element software, such as FEniCSx, eases the implementation of high-order finite element methods. A mass-lumped finite element scheme is implemented, which, combined with the explicit Runge–Kutta method avoids the need to solve a linear system of equations. To further accelerate the solver, a fast finite element integration algorithm is employed. In particular, an algorithm with the lowest known computational cost complexity is implemented, namely the sum-factorisation algorithm. Furthermore, the algorithm is carefully implemented to ensure optimal performance. The solver is implemented for both CPU and GPU architectures. On both architectures, it is shown that the finite element integration implementations achieve a good fraction of peak hardware performance. Additionally, it is shown that the solver exhibits excellent parallel scalability. Contrary to common perceptions, it is demonstrated that a high-order finite element method is well-suited for realistic acoustic simulation for focused ultrasound applications. The novel contributions in this thesis include the development of a high-order finite element method with a fast finite element integration algorithm.

The solver is validated against other acoustic wave solvers for several benchmark problems. Overall, it is shown that the developed solver is fast, accurate, scalable, and is suitable for large-scale linear and nonlinear time-domain acoustic simulations.

For my family, with whom I find my greatest joy.

Acknowledgements

This thesis would not have been possible without the support of many people. I am extremely grateful to my supervisor, Prof. Garth Wells, for his patience and guidance during my PhD. I hope to continue our collaboration in the years to come.

I would also like to thank my research colleagues and friends: Chen, Chris, Igor, Joe, Jørgen, Matt, Nirav, Sam, and Sarah. Your support and help during my research journey have been invaluable.

I extend my gratitude to the Cambridge Malaysian group (Cambridge Circle and Kelab Sukan CambHunt), who not only welcomed me into Cambridge but also provided the support that helped me stay grounded and thrive away from home. Special thanks to Adam & Nadiah, Aiman & Asma, Amin & Atikah, Ashaari & Nadiah, Daeng & Nurul, Fadhli & Sharah, Fadly & Zieyra, Faris & Amirah, Fariz & Nabilah, Fauzi & Huda, Hanafi & Hanis, Helmi & Hidayah, Hisham & Izu, Radhi & Najah, Ramadhan & Hanis, Yati & Dharius, and Yoga & Farah for providing me with the Malaysian hospitality during my time in Cambridge.

I am deeply indebted to my family: Papa, Mama, Ayah, Ibu, Along, Kak Huserk, Angah, Abede, Sepul, Qistina, Shikin, Alip, and my nieces and nephews, for their endless love, encouragement, and support. I present this thesis as a small gift to you all.

Finally, I am extremely grateful to my wife, Nurul Adibah, and my son, Adam Anaqi. Their love, patience, and warmth have made this PhD journey worthwhile.

I would also like to acknowledge the following organisations for their financial support during my time in Cambridge: the Cambridge University Physics Department, the Ministry of Higher Education Malaysia, and the University of Malaya.

*Pisang emas dibawa belayar,
Masak sebiji diatas peti;
Hutang emas boleh dibayar,
Hutang budi dibawa mati.*

Table of contents

List of figures	x
List of tables	xvii
Nomenclature	xviii
1 Introduction	1
1.1 Background	2
1.2 Challenge	3
1.3 Regimes of therapeutic focused ultrasound applications	4
1.4 A review of focused ultrasound simulation	5
1.4.1 Mathematical models for focused ultrasound simulation	6
1.4.2 Numerical methods for focused ultrasound simulation	7
1.5 Objectives	10
1.6 Contributions	11
1.7 Outline	12
2 Mathematical formulation of a focused ultrasound problem	14
2.1 Derivation of the Westervelt equation	14
2.2 Sommerfeld radiation condition	19
2.3 Statement of the initial-boundary value problem	19
2.4 Variational formulation of the initial-boundary value problem	20
2.5 Summary	22
3 Numerical formulation of a focused ultrasound problem	23
3.1 Galerkin approximation of the variational form	23
3.2 Fully discrete formulation using the explicit Runge–Kutta method	24
3.3 Mass-lumped finite element method	27
3.4 Degree-of-freedom map for high-order basis	29
3.5 Solver algorithm	34
3.6 Summary	36

4	Implementation of the numerical methods	37
4.1	FEniCSx software	37
4.2	Distributed memory parallelism	38
4.2.1	Scatter forward	40
4.2.2	Scatter reverse	40
4.3	Implementation overview	40
4.4	The UFL problem description	43
4.5	The FEniCSx-FUS solver	46
4.5.1	Assembly of the global diagonal mass <i>matrix</i>	46
4.5.2	Evaluation of the Runge–Kutta time-derivative functions	47
4.6	Identifying solver bottleneck	48
4.7	Summary	52
5	Fast finite element integration	53
5.1	Cell-wise finite element assembly	53
5.1.1	Evaluation of the local mass operator	53
5.1.2	Evaluation of the local stiffness operator	54
5.2	Precomputing geometric data	55
5.3	Sum-factorisation algorithm and implementation	56
5.3.1	Standard method	56
5.3.2	Sum-factorisation algorithm	58
5.3.3	Tensor contraction through matrix-matrix multiplication	60
5.4	Reduced precision	63
5.5	Performance analysis	64
5.5.1	Precomputing geometric data	64
5.5.2	Sum-factorisation implementation	65
5.5.3	Reduced precision	68
5.5.4	Strong scaling	68
5.6	Summary	70
6	Validation of the FEniCSx-FUS solver	71
6.1	Source windowing	71
6.2	Transcranial focused ultrasound	74
6.2.1	Model equation	74
6.2.2	Transducer configurations	74
6.2.3	Material parameters	74
6.2.4	Simulation setup	76
6.2.5	Post-processing methodology	76
6.2.6	Comparison metrics	77

6.2.7	Mesh convergence study	77
6.2.8	Benchmarks 1 and 2	78
6.2.9	Benchmark 3	87
6.2.10	Benchmark 4	90
6.2.11	Benchmark 5	93
6.2.12	Benchmark 6	96
6.2.13	Benchmark 7	99
6.2.14	Summary of transcranial focused ultrasound simulations	99
6.3	High-intensity focused ultrasound	104
6.3.1	Nonlinear wave propagation in 1D	104
6.3.2	Simulation of nonlinear focused ultrasound application in 3D	106
6.3.3	Summary for HIFU simulations	110
6.4	Summary	110
7	GPU accelerated solver	113
7.1	Difference between a CPU and GPU architecture	113
7.2	Basics of a CUDA kernel implementation	114
7.3	Operator kernels	115
7.3.1	Mass operator kernel	115
7.3.2	Stiffness operator kernel	116
7.4	Multi-GPU implementation	118
7.4.1	Scatter forward implementation	120
7.4.2	Scatter reverse implementation	122
7.5	Performance analysis	123
7.5.1	Operator kernels performance	123
7.5.2	Multi-GPU performance	126
7.6	Simulation examples	126
7.6.1	Linear example	128
7.6.2	Nonlinear example	129
7.7	Summary	129
8	Conclusion and future research	131
8.1	Future research	133
	References	135
	Appendix A Packing and unpacking CUDA kernels	148
A.1	Scatter forward	148
A.1.1	Packing	148
A.1.2	Unpacking	149

A.2	Scatter reverse	149
A.2.1	Packing	149
A.2.2	Unpacking	150

List of figures

1.1	A focused ultrasound setup. The symbol R , f , and Ω denotes the transducer radius, transducer focal length, and the computational domain, respectively. The blue region represents the computational domain, the red region represents the liver, and the brown region represents the tumour.	4
3.1	Examples of the Lagrange Q_l finite element on a quadrilateral with nodes at Gauss–Lobatto–Legendre (GLL) points.	28
3.2	Examples of the Lagrange Q_l finite element on a hexahedron with nodes at Gauss–Lobatto–Legendre (GLL) points.	28
3.3	Degree-of-freedom layout for degree 3 Lagrange element on the reference element. .	30
3.4	A mesh and its finite element space construction from a reference cell.	31
3.5	Constructing the degree 3 Lagrange finite element space. Observed the mismatch of the global degrees-of-freedom l_{10} and l_{11}	32
3.6	The degrees-of-freedom layout obtained after degrees-of-freedom transformation. .	35
4.1	The components of the FEniCSx open-source software.	38
4.2	Illustration of the owned and ghost degrees-of-freedom. When running in parallel, DOF 0 to 19 is send to process 0 and DOF 18 to DOF N is send to process 1, where N is the global number of degrees-of-freedom. Both processes contain DOF 18 and 19. However, DOF 18 and 19 in process 0 are <i>ghost</i> DOF (in red), while process 1 owns DOF 18 and 19 (in blue).	39
4.3	The scatter forward mechanism. In this instance, process 0 owns DOF 13, 14, and 35 (shown in blue). Process 1 contains DOF 13 and 14 as ghost DOF and process 2 contains DOF 35 as ghost DOF (shown in red). The scatter forward operation send data from owned DOF to ghost DOF. In this regard, the data from DOF 13, 14, and 35 in process 0 are send to DOF 13 and 14 in process 1 and DOF 35 in process 2. The direction of the data transfer is represented by the arrows.	41

4.4	The scatter reverse mechanism. In this example, process 0 owns DOF 13, 14, and 35 (shown in blue). Process 1 contains DOF 13 and 14 as ghost DOF and process 2 contains DOF 35 as ghost DOF (shown in red). The scatter reverse operation send data from ghost DOF to owned DOF. In this regard, the data from the ghost DOF 13 and 14 of process 1 and ghost DOF 35 from process 2 is send to process 0. The direction of the data transfer is represented by the arrows.	42
4.5	Visualisation of a FFCx compilation for <code>forms.py</code> and the generated code that is used by <code>Westervelt.hpp</code>	43
4.6	The percentage of time spent within each functions in listing 4.4. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using <i>P4</i> Lagrange element.	50
4.7	The percentage of time spent in the <code>fv</code> function of listing 4.3. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using <i>P4</i> Lagrange element.	50
4.8	The percentage of time spent for vector assembly of each terms in the forms definition of listing 4.1. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using <i>P4</i> Lagrange element. The total time spent is 0.76 seconds. The orange bars represent the terms on the left-hand side and the blue bars represent the terms on the right-hand side.	51
5.1	The comparison between the theoretical computational cost of the standard and sum-factorisation algorithm for the assembly of finite element vector. Notice that as the polynomial order increases, the cost difference between the two algorithms becomes more evident. The dashed lines represent the gradients of the polynomial order. . . .	60
5.2	The percentage of time spent for vector assembly with precomputation of geometric data. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using <i>P4</i> Lagrange element. The total time for assembly is 0.55 seconds. The orange bars represent the terms on the left-hand side and the blue bars represent the terms on the right-hand side.	64
5.3	A comparison between FEniCSx assembly and the assembly with precomputation of geometric data. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using <i>P4</i> Lagrange element. The blue bars represent the FEniCSx implementation and the orange bars represent the precomputation of geometric data implementation.	65

5.4	A comparison between 3 different implementation, namely FEniCSx assembly, the precomputed geometric (PC) assembly, and the sum factorisation assembly. The blue bar represents FEniCSx implementation, the orange bar represents the precomputed geometric data implementation, and the green bar represents the sum-factorisation implementation. This test was performed using the Intel Icelake CPU on 76 processes with 60×10^6 degrees-of-freedom with $P4$ Lagrange element using double floating-point precision.	66
5.5	A comparison between the vector assembly time using the implementation in this work against the Fastor implementation. The blue bar represents the implementation in this work and the orange bar represents Fastor implementation. This test was performed using the Intel Icelake CPU on 76 processes with 60×10^6 degrees-of-freedom with $P4$ Lagrange element using double floating-point precision.	67
5.6	Performance of the vector assembly of the mass operator and the stiffness operator for different polynomial order. The test was conducted on double precision floating-point number using Intel Icelake CPU with 76 processes. The dashed line represents the peak performance of the architecture for triad benchmark. The blue line and orange line represent the performance achieved by the assembly implementation. Blue represents the mass operator and orange represents the stiffness operator.	68
5.7	A comparison between the vector assembly time using double and single precision floating-point numbers. The blue bar represents double precision and the orange bar represents single precision. This test was performed on Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom.	69
5.8	Performance of the mass operator and the stiffness operator for different polynomial orders. The test was conducted using single precision floating-point numbers on Intel Icelake CPU with 76 processes. The dashed line represents the peak performance of the architecture for triad benchmark. The blue line and orange line represent the performance achieved by the assembly implementation. Blue represents the mass operator and orange represents the stiffness operator.	69
5.9	Strong scaling analysis of the FEniCSx-FUS solver on Intel Icelake CPU for problem sizes of 31 million (blue), 62 million (orange), and 126 million (green) degrees-of-freedom, respectively. The dashed line represents perfect strong scaling. The computational experiment was performed using between 2305 and 3697 time steps for 4th order explicit Runge–Kutta method. The data shows the run time per step.	70
6.1	Source function without and with windowing.	72
6.2	Oscillating wall. The source boundary condition and absorbing boundary condition are prescribed on Γ_1 and Γ_2 , respectively. On Γ_3 , a zero Neumann boundary condition is prescribed. This setup ensures that the plane wave propagates to the right and in a single direction.	73

6.3	The rate of convergence of the finite element solution for the plane wave propagation problem. The symbols p and p_h denote the analytical solution and the finite element solution of the pressure field, respectively.	73
6.4	Transducers configuration.	75
6.5	Example meshes used in the simulation.	76
6.6	Source locations and cross-sectional view for the scattering by a sphere problem. The x -axis is set as the axial position, the y and z -axis are set as the lateral position. The axial length is 9 m and the lateral length are 6 m. The radius of the circular planar piston and the spherical scatterer is 1 m.	78
6.7	Convergence of the L^2 and L^∞ relative differences for increasing number of elements per wavelength. The L^2 and L^∞ relative differences are based on table 6.2.	79
6.8	A comparison of the pressure amplitude between OptimUS (left) and FEniCSx-FUS (right) solver. The comparison domain size is 6×8 m 2D slice with 250×250 grid points. Observed that above 20000 elements, the output of both solver is visually similar.	81
6.9	Source locations and cross-sectional view for benchmark 1 and 2. The x -axis is set as the axial position, the y and z -axis are set as the lateral position. The axial length is 120 mm and the lateral length are 70 mm. Reproduced from [16].	82
6.10	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 1 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	83
6.11	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 1 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	84
6.12	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 2 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	85
6.13	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 2 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	86
6.14	Layout for benchmark 3. Reproduced from [16]. The material properties used are given in table 6.1.	87
6.15	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 3 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	88
6.16	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 3 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	89

6.17	Layout for benchmark 4. Reproduced from [16]. The material properties used are given in table 6.1.	90
6.18	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 4 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	91
6.19	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 4 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	92
6.20	Layout for benchmark 5. Reproduced from [16]. The material properties used are given in table 6.1.	93
6.21	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 5 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	94
6.22	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 5 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	95
6.23	Layout for benchmark 6. Reproduced from [16]. The material properties used are given in table 6.1.	96
6.24	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 6 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	97
6.25	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 6 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.	98
6.26	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 7 with focused bowl source. The comparison domain size is $120 \times 70 \times 70$ mm with 0.5 mm grid sampling, which corresponds to $241 \times 141 \times 141$ grid points.	101
6.27	The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 7 with planar source. The comparison domain size is $120 \times 70 \times 70$ mm with 0.5 mm grid sampling, which corresponds to $241 \times 141 \times 141$ grid points.	103
6.28	The comparison between the analytical solution and the finite element solution for the Burgers equation using a 6th degree basis function with mesh size of 6 elements per wavelength. This study is inspired by the numerical experiment performed in [163].	105
6.29	Least square error in percentage for different polynomial order of basis function. The least square error is computed using the harmonic amplitudes, up to the 12th harmonics. The symbols F_m and F_f denote the harmonics based on analytical solution and the finite element solution, respectively.	106

6.30	The comparison between the FEniCSx-FUS solver with the reference solvers for H131 transducer in water. The comparison domain size is 80×40 mm 2D slice with 0.2 mm grid sampling, which corresponds to 357×179 grid points.	109
6.31	The comparison between the FEniCSx-FUS solver with the reference solvers for H101 transducer in liver. The comparison domain size is 120×64 mm 2D slice with 0.2 mm grid sampling, which corresponds to 536×287 grid points.	111
7.1	A schematic comparison between CPU and GPU architectures. Schematic reproduced from [5].	114
7.2	The normalised assembly times for the operator kernels. The symbols Γ and Ω denote integration over facets and cells, respectively. This was profiled on the NVIDIA RTX A4000 GPU with 2.1×10^6 degrees-of-freedom using a $P4$ continuous Lagrange finite element. The absolute time for each operator are 0.16×10^{-3} s for the mass operator on the boundary, 0.40×10^{-3} s for the mass operator on the cell, and 1.5×10^{-3} s for the stiffness operator on the cell.	124
7.3	The roofline plot for the stiffness kernel between the 64-bit (circle with red edge) and 32-bit (circle with blue edge) floating-point precision. This data was collected by running an experiment with 33×10^6 degrees-of-freedom using a $P4$ continuous Lagrange finite element. The experiment was performed on a single NVIDIA RTX A4000 GPU.	125
7.4	The roofline plot for the stiffness kernel between the Basix (circle with red edge) and tensor product (circle with blue edge) order. This data was collected by running an experiment with 33×10^6 degrees-of-freedom using $P4$ continuous Lagrange finite element. The experiment was performed on a single NVIDIA RTX A4000 GPU.	125
7.5	The roofline plot for the stiffness kernel for different polynomial order. The edge color of the circle represents different polynomial order. In increasing order, the green edge represent $P3$, the violet edge represent $P4$, the orange edge represent $P5$, the red edge represent $P6$, and the blue edge represent $P7$. The basis function is of tensor product ordering. The experiment was performed on a single NVIDIA RTX A4000 GPU.	126
7.6	The Nsight Systems report of the MPI communication. The Memory PtoP denotes the peer-to-peer communication between the <i>devices</i> . This experiment was performed using MPI on a connection of 4 A100 NVIDIA GPU.	127
7.7	Strong scaling analysis of the FEniCSx-FUS GPU solver on NVIDIA DGX A100 platform for problem size of 430 million degrees-of-freedom. The dashed grey line represents perfect scaling and the blue line represents scaling achieved. The experiment was performed using 4th order explicit Runge–Kutta method with 34300 time steps. The data shows the run time per step.	127

7.8	The comparison between the CPU and GPU implementation of the FEniCSx-FUS solver for benchmark 1 with planar source.	128
7.9	The comparison between the CPU and GPU implementation of the FEniCSx-FUS solver for H131 transducer at 100W in water.	130

List of tables

3.1	Butcher-tableau for the classical 4th order explicit Runge–Kutta scheme.	27
6.1	Material parameters used in the simulations.	75
6.2	Difference metrics used for the comparison between the FEniCSx-FUS solver with the k-Wave solver. The symbols p_f and p_k represent the pressure amplitude field for the FEniCSx-FUS solver and the k-Wave solver sampled at specific grid points, respectively. Sum and maximum values are assumed to be over all values in the comparison domain, starting from the exit plane of the transducers [16]. The function pos max denotes the position of the maximum pressure amplitude.	77
6.3	The operating frequencies and geometrical data of the focused bowl transducers, taken from [4]. The geometrical parameters are the focal length l and the transducer radius R . Table reproduced from [67].	107
6.4	Material parameters for used in the numerical experiments. Obtained from [17, 51].	108

Nomenclature

Greek Symbols

$\phi(x)$ Finite element basis function

Γ Boundary

μ_B Bulk viscosity

ρ Medium density

δ Diffusivity of sound

Ω Domain

δ_{ij} Kronecker delta

β Coefficient of nonlinearity

α_q Quadrature weights

μ Shear viscosity

κ Thermal conductivity

Superscripts

n Time step

Subscripts

h Finite element solution

Other Symbols

K Elements

S_h Finite dimensional space

R Gas constant

N	Global degrees-of-freedom
H^1	Hilbert space
J_K	Jacobian matrix
L	Second-order Lagrangian density
N_c	Number of elements
N_q	Number of quadrature points
P	Thermodynamic pressure
p	Pressure field
x_q	Quadrature points
$g(t)$	Source function
s	Specific entropy per unit mass
c_v	Specific-heat coefficient at constant pressure
c_p	Specific-heat coefficient at constant volume
c_0	Speed of sound
L^2	Square integrable function space
t	Time
$\mathbf{u}$	Velocity vector
u_i	Component of velocity vector

Acronyms / Abbreviations

API	Application Programming Interface
ASA	Angular spectrum approach
BEM	Boundary element method
BVP	Boundary value problem
CPU	Central processing unit
DOF	Degrees-of-freedom

FDTD	Finite-difference time-domain
FEM	Finite element method
FFCx	FEniCS Form Compiler
FFT	Fast Fourier transform
FUS	Focused ultrasound
GLL	Gauss-Lobatto-Legendre
GPU	Graphics processing unit
HAS	Hybrid angular spectrum
HIFU	High-intensity focused ultrasound
IBVP	Initial-boundary value problem
IFFT	Inverse fast Fourier transform
INCS	Iterative nonlinear contrast source
KZK	Khokhlov-Zabolotskaya-Kutznesov
MPI	Message Passing Interface
PDE	Partial differential equation
SEM	Spectral element method
SIMD	Single-instruction, multiple-data
tFUS	Transcranial focused ultrasound
UFL	Unified Form Language

Chapter 1

Introduction

Focused ultrasound (FUS) is a non-invasive medical therapy that uses ultrasound to heat tissue and perform ablation [97]. Acoustic waves generated by transducers are used to heat small tissue regions while leaving the surrounding tissue unaffected. The non-invasive nature of FUS makes it an attractive treatment option because it reduces the risk of bacterial infection [154]. In contrast to chemotherapy, FUS is non-ionising and non-toxic, which enables multiple repeated treatments. Moreover, FUS treatment requires minimal to no anaesthesia, thereby lowering costs and avoiding unwanted side effects [154]. Importantly, the reduced need for anaesthesia allows patients to quickly resume their daily activities. FUS shows promising potential for the treatment of prostate and liver cancer [40, 183]. FUS has also been used for treating brain diseases such as Parkinson's disease [116, 108] and for controlling the opening of the blood-brain barrier [41, 113]. In cosmetics, it is used for the tightening of the face skin [122].

The mechanisms of FUS are primarily mechanical and thermal [41], and depend on the intensity, frequency, and duration of the ultrasound waves. In terms of the mechanical effect, the ultrasound waves generate localised mechanical forces due to the propagation of the sound waves. As a result, it induces mechanical effects such as microstreaming and cavitation, which can enhance drug-delivery by opening the blood-brain barrier [116]. With regard to the thermal effect, ultrasound waves absorbed by tissues produce heat due to the conversion of mechanical energy into thermal energy. At sufficiently high intensities, this results in significant temperature elevations, a mechanism utilised by high-intensity focused ultrasound (HIFU) to ablate tissue in applications such as cancer treatment [193] or uterine fibroid removal [88].

Even though FUS treatment has appealing features, its adoption as a treatment modality remains low due to the difficulty in designing effective patient-specific treatment plans [87]. Achieving precise heating of the targeted area without affecting the surrounding tissue is a challenge in practice. For instance, poor placement of FUS transducers may cause skin burns in patients [45]. Additionally, obstructions such as bones and internal organs, can reduce the effectiveness of ultrasound waves [87, 121]. A barrier to the use of simulation for treatment planning is the time required for FUS simulation [109]. A lengthy simulation time renders patient-specific treatment plans impractical,

with medical practitioners unable to test different scenarios. Therefore, a fast and high-fidelity FUS simulation is a promising tool to enable effective patient-specific treatment planning.

1.1 Background

In this thesis, the development of a solver for acoustic wave propagation problem with a view to FUS applications is presented. Such a solver can also support a wide range of other applications. For example, seismic wave propagation for the detection of oil reservoirs [55], room acoustics [126], and sonar research [77]. This thesis focuses on a specific application, namely the propagation of FUS waves within the human body.

Several numerical methods have been used for acoustic wave propagation problems, including the finite-difference time-domain (FDTD) method [104], the k -space pseudospectral method [42] and the boundary element method (BEM) [75]. A the fundamental issue with FDTD and k -space pseudospectral methods are that both methods suffer from the staircasing effect when approximating general boundaries or material interfaces [136, 71]. This is due to the spatial discretisation, which requires the geometry to be discretised using structured grid. While several techniques exist to mitigate this limitation [157, 182], the implementation of such techniques is not trivial. In contrast to FDTD, BEM uses a conforming spatial discretisation of the boundary and, as such, does not suffers from the staircasing effect. In addition, BEM's solution is naturally free from pollution error—error due to numerical dissipation and dispersion. However, even though BEM can give an accurate solution, the resulting matrix due to the BEM formulation is dense; thus, BEM incurs a large computational cost, especially for 3D acoustic problems. Additionally, implementing BEM for inhomogeneous problems is challenging.

Like BEM, the *finite element method* (FEM) solves boundary value problems (BVP) by discretising the governing partial differential equation (PDE). FEM uses linear combinations of piecewise polynomial basis function to approximate the solution field. A subset of FEM in which the basis function is of high-order is called the *spectral element method* (SEM) [188]. SEM uses Gauss-Lobatto-Legendre (GLL) interpolation points rather than equispaced points to define the *nodal* piecewise polynomial basis functions. SEM possesses excellent convergence properties with fewer degrees-of-freedom [188]. Additionally, in combination with an appropriate quadrature scheme, SEM can significantly reduce the computational cost, especially for time-dependent problems [12]. Moreover, for tensor product elements—quadrilateral and hexahedral elements—there are algorithms available for fast numerical quadrature evaluation such as the sum factorisation algorithm [114, 28, 94].

SEM has been widely used in computational fluid dynamics and seismic wave propagation [94, 101, 102]. However, for high-frequency wave problems such as in medical ultrasound applications, the usage of SEM remains limited. In fact, in a recent collective effort to model transcranial focused ultrasound problems, only 1 out of the 11 teams used SEM [16]. One notable factor contributing to FEM or SEM's limited popularity in high-frequency wave problems is computational cost. Specifically, the high domain-to-wavelength ratio in such problems can result in significant computational demands.

Nonetheless, with regard to finite and spectral element methods, several efforts have been made to improve the efficiency of 3D numerical simulations. Recently, Africa *et al* [11] developed a numerical solver for cardiac electrophysiology based on the spectral element method. They implemented a matrix-free formulation that achieves a significant speed-up in contrast to the matrix-based formulation. Likewise, Maier and Kronbichler developed a high-performance finite element solver for the computation of 3D compressible Euler equations [110]. The solver uses a novel algorithmic design that allows efficient computation of floating-point operations resulting in excellent parallel scalability.

While advances have been made in cardiac electrophysiology and compressible fluid dynamics applications, this thesis focuses on the nonlinear acoustic wave propagation problem. A reason low order finite element methods are considered less appealing for wave problems is due to the pollution error, especially in high-frequency wave problems. Since the finite element solution depends on the element size, accurately resolving high-frequency waves requires many elements, which in turns increases the computational cost. However, studies have shown that the pollution error can be significantly reduced by using a high-order basis [84]. Moreover, even though high-order bases generate more degrees-of-freedom per cell, the number of cells per wavelength required is smaller. Consequently, using a high-order basis can lower the global number of degrees-of-freedom.

Accordingly, the aim of this thesis is to develop an efficient high-order finite element solver to simulate 3D FUS applications. To this end, a fast algorithm is identified and implemented in the context of the high-order finite element method. Altogether, at each numerical modelling stage, maximum achievable performance is sought, while maintaining acceptable numerical accuracy.

1.2 Challenge

A key challenge in simulating a FUS application is that it involves a 3D, heterogeneous unbounded domain. Firstly, treatment of the unbounded domain is required. Secondly, the handling of a heterogeneous medium is also required. Finally, a 3D simulation of therapeutic ultrasound simulation can be computationally expensive because the number of degrees-of-freedom required is dependent upon the number of wavelengths across the computational domain.

Consider the focused ultrasound setup illustrated in figure 1.1. The symbol R denotes the radius of the source transducer. The focal length is denoted by f and the simulation region is denoted by Ω . In this setting, the bowl-shaped transducer is considered to be in contact with the patient's body via a coupling medium—typically water. Ultrasound waves propagate from the transducer through the coupling medium and a homogeneous region (tissue) before reaching the heterogeneous target region. The aim is to accurately simulate wave propagation in the computational domain and subsequently predict the pressure amplitudes in the target region.

To illustrate the computational needs of an ultrasound simulation, consider a clinical setting where the transducer¹ has an operating frequency of F_0 of 1 MHz, a focal length f of 18 cm and

¹An example of the range of FUS transducer's properties can be obtained from[4]

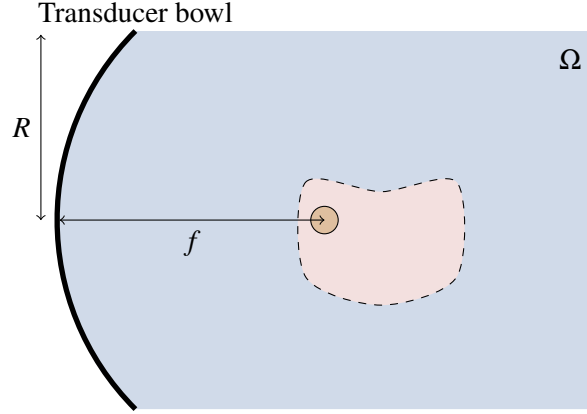


Fig. 1.1 A focused ultrasound setup. The symbol R , f , and Ω denotes the transducer radius, transducer focal length, and the computational domain, respectively. The blue region represents the computational domain, the red region represents the liver, and the brown region represents the tumour.

an outer radius R of 8 cm. Suppose a region with length L of 20 cm is to be simulated, using the speed of sound in water ($c_w = 1480 \text{ m s}^{-1}$), the fundamental wavelength is $\lambda = c_w / F_0 = 1.48 \text{ mm}$. Thus, in terms of the fundamental wavelength, the domain is $2R \times 2R \times L \approx 108\lambda \times 108\lambda \times 135\lambda$. To accurately simulate this setting using a first-order finite element basis, a resolution of $n_{\text{res}} = 10$ elements per wavelength is typically used [84]. Accordingly, simulating the entire domain requires $1080 \times 1080 \times 1350$ elements, which corresponds to approximately 1.6×10^9 degrees-of-freedom. To make matters worse, higher resolution is needed if higher harmonics are of interest, which is typical in high-intensity focused ultrasound problem.

1.3 Regimes of therapeutic focused ultrasound applications

In this section, a brief overview of the regimes of therapeutic FUS applications is presented. The regimes of a therapeutic FUS application are categorised by the biological effects it induces, which, in turn, depend on several factors, such as the intensity and frequency (low or high), the modes of operation (continuous or pulsed), and the total duration of exposure to the ultrasound fields [179]. Unlike diagnostic ultrasound, which is primarily used for imaging, therapeutic ultrasound employs higher acoustic intensities [41]. The bioeffects induced by therapeutic FUS applications can be categorised into histotripsy, hyperthermia, nonthermal effects, and thermal ablation [179].

Histotripsy is driven by the mechanical effects of FUS, primarily through acoustic cavitation [41]. This technique induces irreversible tissue damage by causing liquefaction, wherein mechanical forces disrupt the tissue at the cellular level [192]. Histotripsy operates at high-intensities, between 0.5 to 30 kW cm^{-2} , with short pulse durations between 1 to $10 \mu\text{s}$ [192]. The operating frequency typically ranges from 500 kHz to 2 MHz , with exposure time varying from a few seconds to several minutes [41, 192, 179].

Hyperthermia is a phenomenon in which the temperature of a localised tissue region is increased due to the absorption of ultrasound energy and viscous damping [58, 154]. In contrast to histotripsy, the effects of hyperthermia are reversible. It is used to enhanced drug delivery [117, 186] and radiosensitisation [194]. FUS for hyperthermia typically operates at moderate to high intensities, ranging from 0.1 to 2 kW cm^{-2} , using pulsed mode to avoid irreversible tissue ablation [179]. The operating frequency for hyperthermia ranges from 0.5 to 3 MHz , with exposure times between 10 to 60 minutes [58, 194].

Nonthermal therapeutic FUS mechanisms include tissue displacement [108], microbubble cavitation [13], and acoustic streaming [83]. This modality operates at low to moderate intensities, ranging from 0.03 to 3 W cm^{-2} , with a short pulsed mode to prevent tissue heating [91, 118]. The operating frequency for nonthermal therapeutic FUS is between 0.2 to 3 MHz and exposure times varying from a few seconds to several minutes [91, 179]. Nonthermal therapeutic FUS is used in drug delivery [76, 34] and neuromodulation [29].

Similar to hyperthermia, the physical mechanism of thermal ablation also results from the absorption of ultrasound energy and viscous damping. However, the effect of thermal ablation is irreversible, as it is primarily used to cause permanent tissue damage by raising the tissue temperature above 55°C [154]. Its applications include the treatment of cancers [31, 156] and immunomodulation [134]. The intensity used for thermal ablation is high, ranging from 0.5 to 10 kW cm^{-2} , with a continuous wave mode [192]. FUS for thermal ablation operates at frequency between 0.5 to 5 MHz , with exposure times between 1 to 10 minutes to ensure thermal necrosis [192].

This thesis presents the development of a time-domain FUS solver based on the high-order finite element method. The solver is designed to simulate a range of FUS scenarios that induce the bioeffects discussed in this section. As a time-domain solver, it is theoretically capable of simulating both pulsed and continuous wave operating modes for all FUS applications. However, as will be detailed later in the thesis (see chapter 6), the validations of the solver are conducted solely using the continuous wave mode due to the availability of established benchmarks.

1.4 A review of focused ultrasound simulation

In this section, a brief review on the development of numerical solvers for FUS simulation is provided. In particular, the efforts that have been made to simulate a FUS application are presented. The model equations and the numerical methods that have been previously developed are discussed. It is highlighted that a fast 3D simulation of FUS remains a challenge, especially for nonlinear problems such as high-intensity focused ultrasound simulations; this motivates the development of the numerical solver presented in this thesis. For an extensive review of the mathematical models and the numerical methods for focused ultrasound simulations, the readers can refer to [68].

1.4.1 Mathematical models for focused ultrasound simulation

The model equations that govern wave propagation for focused ultrasound applications are based on the laws of conservation, namely conservation of mass, conservation of momentum, and conservation of energy [74]. Retaining the first-order terms of the conservation equations results in the linear wave equation, that reads

$$\frac{1}{\rho_0} \nabla^2 p - \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} + \frac{\delta}{\rho_0 c_0^2} \nabla^2 \frac{\partial p}{\partial t} = 0, \quad (1.1)$$

where p is the pressure field, t denotes time, and ρ_0 , c_0 and δ denote the density, speed of sound, and the diffusivity of sound of the medium, respectively. The second-order linear wave equation has been used extensively in medical ultrasound applications [16, 109, 121]. Among the model equations considered for medical ultrasound applications, the linear wave equation is the easiest model to solve since there is no nonlinearity. For simple models, such as the propagation of waves in a homogeneous medium due to a vibrating piston, there exist analytical solutions. However, the linear wave equation is not suitable for some focused ultrasound applications, such as for high-intensity focused ultrasound [154, 41]. Nevertheless, the linear wave equation is still widely used to model ultrasound devices operating in low-intensity regimes [16, 68].

The limitation of the linear wave equation motivates the use of more elaborate models, such as the Khokhlov–Zabolotskaya–Kutznosov (KZK) equation. The KZK equation reads [68]

$$\frac{\partial^2 p}{\partial z \partial t'} - \frac{c_0}{2} \nabla_{\perp}^2 p - \frac{\delta}{2c_0^3} \frac{\partial^3 p}{\partial t'^3} - \frac{\beta_0}{2\rho_0 c_0^3} \frac{\partial^2 p^2}{\partial t'^2} = 0, \quad (1.2)$$

where t' is the retarded time ($t' = t - z/c_0$) and $\nabla_{\perp}^2 = \partial^2/\partial x^2 + \partial^2/\partial z^2$ is the transverse Laplacian. The transverse Laplacian captures the spatial variations of a scalar field perpendicular to the direction of wave propagation. The symbol β_0 denotes the coefficient of nonlinearity of the medium. In contrast to the linear wave equation, the KZK equation contains a nonlinear term (fourth term on the left-hand side of equation (1.2)). In fact, the KZK equation consists of a diffraction term, an absorption term, and a nonlinear term. The KZK equation is one of the simplest acoustic models that includes nonlinear effects [148, 191, 189]. The KZK equation is a parabolic approximation of the Westervelt equation. In contrast to the Westervelt equation, the KZK equation is less accurate in the near-field and at position off the main axis [74, 68]. This means that a wide-angled transducer is not accurately model by the KZK equation. In practice, the KZK equation is suitable for a focused transducer with angle between 15° and 16° of the beam axis [155]. However, in the frequency domain, studies have shown that the KZK equation can accurately model a wider transducer angle of about 25° —this is known as wide-angle KZK equation [148]. Another limitation of the KZK equation is that it is a one-way wave propagation model [82], whereby it does not model reflections that can occur due to heterogeneity of a medium.

The Westervelt equation is formulated by retaining the second-order term of the conservation equations (the derivation is presented in section 2.1). The Westervelt equation reads [74, 49]

$$\frac{1}{\rho_0} \nabla^2 p - \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} + \frac{\delta}{\rho_0 c_0^2} \nabla^2 \frac{\partial p}{\partial t} = -\frac{\beta}{\rho_0^2 c_0^4} \frac{\partial^2 p^2}{\partial t^2}. \quad (1.3)$$

Each of the medium parameters (ρ_0 , c_0 , δ , β) in equation (1.3) can be spatially varying. The Westervelt equation is widely used to model high-intensity focused ultrasound applications. The equation takes into account diffraction (first term), attenuation (third term), and nonlinearity (last term). Being a two-way equation, the Westervelt equation is more challenging to solve than the KZK equation, since both the transmitted and the reflected waves need to be considered [190, 162]. The Westervelt equation is known to be accurate even for wide-angle transducers [68, 166].

A key aspect in the modelling of FUS is the mechanism of acoustic attenuation in tissue. Acoustic attenuation occurs due to wave scattering and absorption. Wave scattering happens when the acoustic wave encounters inhomogeneities or obstacles in a medium, which causes the wave energy to be redistributed in directions other than the propagation direction [121, 84]. Absorption is due to the effects of thermoviscous losses and molecular relaxation [61]. Thermoviscous losses refer to the conversion of acoustic energy into heat due to frictional and viscous properties of the medium. In contrast, molecular relaxation occurs when molecules dissipate energy during the relaxation back to equilibrium. In chapter 2, it is shown that acoustic attenuation is modelled by the diffusivity of sound term for a viscous fluid.

1.4.2 Numerical methods for focused ultrasound simulation

The development of numerical methods for the presented wave equations is an ongoing research endeavour. Several methods exist to solve these equations in either a homogeneous or a heterogeneous media. A brief review of the numerical methods developed to simulate focused ultrasound applications is presented. A comprehensive review of the methods can be found in [68, 173].

The angular spectrum approach (ASA) is a method whereby the pressure field on a desired plane is computed based on the field on a known plane—typically the source plane [64, 37]. In particular, the method involves the computation of the propagation of plane waves in the frequency domain. This is achieved through the 2D fast Fourier transform (FFT) algorithm. The Fourier components of each plane wave are propagated separately and summed using the inverse fast Fourier transform (IFFT) algorithm. Usage of the fast Fourier algorithms is the basis for the efficiency and accuracy of the ASA method. To achieve the focusing effect of a focused transducer, ASA method uses phased arrays, with the phase of the sources set up to generate a focusing effect. Consequently, this is one of the inherent limitations of ASA for focused ultrasound simulation. However, there are efforts to model curved transducers using the ASA method [184, 175]. Since it was initially developed to handle homogeneous media, the accuracy of the ASA method suffers when used for a high contrast medium. However, for weak contrast problems, an acceptable accuracy can be obtained [36]. An improvement to the ASA

method to handle heterogeneous media was developed by Vyas and Christensen [174]. The method is known as the hybrid angular spectrum (HAS) method. To account for the heterogeneity, the method uses an iterative procedure between the space domain and the spatial-frequency domain. Additionally, reflections due to heterogeneity are modelled using a first-order reflections approximation. HAS has been used to simulate transcranial focused ultrasound applications [109, 16]. It was observed that HAS solutions differ from the reference solution, which may be attributed to the first-order reflection approximation [16]. For nonlinear problems, a modified ASA method was developed [90]. The modified ASA method computes an implicit solution to the nonlinear term using a Green's function. An improvement to the modified ASA method was implemented by Jing *et al*; good agreement between the simulation and experimental results was observed [89].

A popular method for nonlinear problems, either the KZK or Westervelt model, in homogeneous media is the operator-splitting method—also known as the split-step method [173]. In this method, the effects of diffraction, attenuation, and nonlinearity are considered separately and are computed independently. More precisely, consider a time step Δt , the solution at $t + \Delta t$ is computed sequentially for each of the effect. One of the main operator-splitting methods is the Strang splitting, which is second-order accurate [153]. Instead of taking Δt step, the Strang splitting methods takes a $\Delta t/2$ step. The operator-splitting technique is frequently used to solve KZK equation. The *Texas* code solved the KZK equation using the operator-splitting method in the time domain [107, 39]. On the other hand, the *Bergen* code solved the KZK equation in the frequency domain using the operator-splitting method [8, 19]. More recently, Yuldashev and Khoklova uses the operator-splitting technique to simulate a 3D Westervelt equation problem [190]. The diffraction operator was solved using the ASA method and the attenuation and nonlinear operator are solved using a hybrid method. The method uses a combination of the frequency-domain method together with the conservative Godunov-type scheme, which enables the method to capture shock formation. While operator-splitting methods have been popular for modelling nonlinear ultrasound propagation [44, 99, 172, 130], the methods inevitably introduces errors since the effects of diffraction, attenuation, and nonlinearity are in fact interdependent.

The iterative nonlinear contrast source (INCS) method was introduced by Huijssen and Veirweij [80, 81]. An extension to the INCS method to handle heterogeneity was introduced in [47, 48]. The INCS method treats the nonlinear term of the Westervelt equation as a contrast source. More precisely, the method solves the linear wave equation by treating the nonlinear term as a source function. The total solution is then computed iteratively using the Neumann iterative solution scheme. The INCS method was developed for medical diagnostic applications and as such is typically used for the simulation of moderately nonlinear problems. A limitation of the method is that it uses the four-dimensional Fourier transform, which leads to large memory requirements, especially for 3D problems.

In contrast to the ASA method, operator-splitting, and INCS methods, several methods have also been developed to solve the full-wave model. In general, these methods involve the discretisation of the domain into a grid. The finite-difference time-domain (FDTD) method divides the domain

into evenly distributed grid points and computes the solution at grid points. The FDTD method is popular for ultrasound simulation due to the simplicity of the formulation and implementation. Hallaj and Cleveland implemented a fourth-order spatial and second-order temporal FDTD discretisation of the Westervelt equation to demonstrate the nonlinear effect on the heating of tissue-like material [73]. In [63], Ginter *et al.* use a high-order FDTD scheme to solve for the full-wave equation, similar to [73], they show that the linear model underestimates the pressure amplitude of the focus. Pinton *et al.* implemented an FDTD method for 3D Westervelt equation with arbitrary frequency dependent attenuation and variations in density [129]. In [50], Doinikov *et al.* implemented the FDTD method to validate their numerical model against experimental data. They found that the Westervelt equation provides good agreement between simulated and measured data of small-amplitude and finite-amplitude excitations problems. In particular, they show that the Westervelt equation is suited for modelling phased array transducers. While FDTD method is simple to formulate and implement, the main drawback of the FDTD method is that it requires many grid points per wavelength, especially for high-frequency problems. Moreover, the method does not handle complex geometries naturally, which consequently causes staircasing effect especially at the interface between different materials [71, 111]. The method can also suffer from phase error due to finite difference approximation of the spatial derivatives [173].

In contrast to the FDTD method, for a homogeneous problem, the k -space pseudospectral method eliminates phase errors [26, 27]. The k -space pseudospectral method eliminates the phase error by introducing corrections to the spatial derivatives [173]. However, it is important to note that for heterogeneous problems, the phase error cannot be completely eliminated [163]. The k -space pseudospectral method is typically used to discretise coupled continuum equations. Tabei *et al.* implemented the method for coupled first-order acoustic wave propagation equations [151]. More recently, the k -space pseudospectral method has been used to solve an extended version of the continuum equations that account for heterogeneity, attenuation, and nonlinearity [161, 160, 163]. The k -space pseudospectral method has been very successful for the simulation of nonlinear ultrasound applications, especially for weak to moderate nonlinearities. However, one of the limitations of the method is that, due to the use of 3D FFT, it requires a global all-to-all parallel data exchange that may become a computational bottleneck especially when deploying the code on computer clusters [86]. Additionally, the method requires large amount of memory, especially for 3D nonlinear HIFU problems [86].

More recently, the boundary element method (BEM) has been used to model ultrasound applications [121, 167, 169, 75, 170, 171]. The boundary element method subdivides the boundary regions of the scatterer into elements—surfaces in 3D and edges in 2D. The acoustic field solution is then represented using a linear combination of basis functions. The main advantage of BEM is that it reduces the dimensionality of the problem since only the scatterer boundary needs to be discretised [173]. Additionally, it removes the need to truncate the computational domain. However, a limitation of BEM is that it does not handle nested or multiple scatterers naturally. Even though efforts have been made to overcome this limitation [75], it remains a challenge, especially for high contrast media.

Also, BEM results in a dense matrix. As a result, running a BEM solver requires large amount of memory as well as good linear solvers. Nonetheless, much research has been done to address these issues especially for FUS applications [167, 75, 23]. Another limitation of BEM is the reconstruction of the acoustic field solution in the post-processing stage is also computationally expensive. However, this limitation can be addressed through MPI implementations, whereby each rank can handle a partition of the mesh. BEM is also known to be difficult for nonlinear problems [185].

Like BEM, FEM represents the acoustic field solution using a linear combination of basis functions. However, in contrast to BEM, FEM subdivides the domain into a collection of elements—known as the mesh. Similar to BEM, FEM is capable of accurately representing complex scatterers and source geometries. In modelling ultrasound applications, Tsuchiya and Kagawa found that FEM is suited for modelling a focusing source for weakly nonlinear acoustic simulation [166]. Hoffelner *et al* developed and implemented FEM for the Kuznetsov equation [79]. They found good agreement between the numerical and the analytical solutions. In [176], Walsh and Torres formulated and implemented FEM for 3D nonlinear acoustic problem in a dissipating fluid. They demonstrate that the time step size necessary to resolve wave distortion decreases as steepening occurs. It should be noted that these FEM studies used low-order finite elements.

Although FEM has been shown to be suitable for modelling focused ultrasound applications [166, 79, 176], the adoption of the method for acoustic wave problems is still limited. This can be attributed to a perception that FEM is computationally inefficient for acoustic wave propagation [173, 68]. This conception is justifiable since early implementations of FEM were based on low-order polynomial basis functions [133], thus requiring many elements per wavelength to resolve the acoustic wave. However, high-order polynomial basis functions require fewer elements per wavelength due to their effectiveness in resolving the acoustic wave profile. Furthermore, the parallelism provided by modern computer architectures, such as GPUs, is suited to the use of high-order basis functions [9, 100]. Hence, the aim is to dispel the perception of the computational inefficiency of FEM for nonlinear acoustic simulation by developing a high-order finite element method that uses efficient algorithms and leverages the capabilities of modern computer architectures, instructions, and parallelism.

1.5 Objectives

This thesis aims to develop a *fast* and *accurate* high-order finite element solver for focused ultrasound applications. As presented in section 1.4, several numerical methods have been used for FUS applications. Accordingly, a fair comparison of speed between different solvers is challenging. Furthermore, the strengths and limitations of each method and implementation are highly dependent on the FUS application [159]. Thus, in this work, a *fast* implementation is defined as a method with low algorithmic complexity and a high fraction of peak performance. The aim is to develop a solver that can achieve a good fraction of peak performance on modern high performance computing hardware. Additionally, the aim is to build a solver that has good parallel scalability. In terms of

accuracy, it is desired that the solver provides solutions that are comparable to other established FUS solvers. In this regard, for the linear model, the solver is validated using several benchmark problems [16], and for the nonlinear model, the solver is compared with the HITU [148] and VINES [66] solvers.

Based on a review of established solvers [160, 43, 148, 69, 10], a focused ultrasound solver is expected to have the following features:

- The solver can model common focused ultrasound transducer geometries, namely planar and bowl geometries.
- The solver can model heterogeneous media.
- The solver can model complex scatterer geometries.
- The solver can model acoustic attenuation and nonlinearity.

Based on the requirements above, the objectives of this thesis are as follows:

- Formulation of the variational problem for FUS applications based on the Westervelt equation for a heterogeneous media.
- Formulation and implementation of the high-order finite element for spatial discretisation and fourth-order explicit Runge–Kutta method for temporal discretisation.
- Implementations of the sum-factorisation algorithm for *fast* finite element integration on CPU and GPU architectures.
- Implementations with different floating-point precisions.
- Performance analysis of the implementations.
- Numerical validation of the solvers.

1.6 Contributions

This thesis concerns the development of *fast* and accurate numerical acoustics solver for FUS applications. In particular, to the development of a *fast* FUS solver based on high-order finite element method.

It is well known in the FUS community that the simulation of FUS applications is computationally challenging, especially high-intensity focused ultrasound in heterogeneous domains. To address this challenge, the use of high-order finite element methods on tensor-product based cells is proposed. These design choices enable the implementation of fast finite element integration algorithms as well as circumventing the assembly of a matrix, thereby reducing the memory requirements, a novelty with

regard to FUS solvers. Besides, in conjunction with an explicit time-stepping algorithm, the need to solve a linear system is avoided.

The developed solver also provides further advantages owing to the finite element method. Firstly, the solver is not restricted to Cartesian grids, providing greater flexibility in modelling the geometric complexity of scatterers. Additionally, the solver can naturally handle heterogeneity within a domain, enabling the incorporation of medium variations within specific organs.

The solver is designed to handle arbitrarily high-order Lagrange finite element basis functions, also a key novelty for a FUS solver. This thesis contributions also include studies on the performance of the FUS solver, which highlight that the solver achieves a good fraction of the theoretical peak performance on modern high-performance computing architectures. Moreover, the solver shows good parallel strong scaling for both CPU and GPU architectures. Overall, the implementation shows that high-order finite element methods are suited for realistic time-domain acoustic simulations. The solver is free and open-source. The CPU and GPU implementations of the solver are available from FEniCSx-FUS and FEniCSx-FUS-GPU repositories, respectively.

The work in this thesis has been presented at the following conferences:

1. **Fast finite element solver for focused ultrasound applications**, presented at SIAM Computational Science and Engineering 2023, Amsterdam, 2023.
2. **Leveraging DOLFINx data-oriented design for GPU implementation**, presented at FEniCS 2024, Oslo, 2024.

1.7 Outline

In chapter 2, the derivation of the initial-boundary value problem used in this work is briefly presented. Specifically, the derivation of the Westervelt equation. Additionally, the Sommerfeld radiation condition that provides the basis of the absorbing boundary condition is presented. The chapter ends with the variational formulation of the initial–boundary value problem.

In chapter 3, the numerical formulation of the initial-boundary value problem is outlined. To this end, the mass lumped finite element method for the spatial discretisation and the Runge–Kutta method for the temporal discretisation are employed. The combination of these two methods are the basis of the numerical solver. Additionally, the degree-of-freedom map for high-order basis, which is important for quadrilateral and hexahedral meshes is discussed. A summary of the solver algorithm is also presented.

In chapter 4, the CPU implementation of the FEniCSx-FUS solver is presented. In particular, the components of the CPU implementation are presented, including an introduction to the FEniCSx software and the distributed memory parallelism of DOLFINx. In addition, the key implementations of the FEniCSx-FUS solver are presented and computational bottlenecks within the solver are identified.

In chapter 5, the methods to accelerate the performance of the FEniCSx-FUS solver are presented. In particular, fast finite element integration, which includes the precomputation of geometric data and

the sum-factorisation algorithm are presented. Furthermore, a presentation of the different floating-point precision feature of the solver is provided. Finally, computational experiments to validate the performance of the implementations are presented.

In chapter 6, the outputs of the FEniCSx-FUS solver are validated. This includes a convergence study to justify the use of the windowing function on the source boundary condition. Additionally, to demonstrate that the solver is suitable for realistic 3D simulations, comparisons are made with established solvers used in transcranial and high-intensity focused ultrasound simulations.

In chapter 7, the FEniCSx-FUS solver is developed for multi-GPU support. In particular, the implementation of the mass and stiffness operator as GPU kernels is presented. To enable multi-GPU utilisations, the MPI communicators are implemented. Performance analyses are performed on the GPU implementation to ensure that it is optimal. Finally, the outputs of the GPU implementation are validated against the CPU implementation.

In chapter 8, a summary of contributions and suggestions for future development of the solver is presented.

Chapter 2

Mathematical formulation of a focused ultrasound problem

Focused ultrasound modelling is based on conservation equations and the equation of state [74]. More precisely, the acoustic equation relevant for focused ultrasound simulation is derived from approximations of the conservation equations and the equation of state. In this chapter, the initial-boundary value problem (IBVP) is derived and the variational formulation used in this thesis is presented.

In section 2.1, a single wave equation—the Westervelt equation—is derived from the conservation equations and the equation of state. In section 2.2, to ensure the exterior IBVP is well-posed, the Sommerfeld radiation condition is introduced. Conveniently, the Sommerfeld radiation condition is equivalent to a first-order absorbing boundary condition. In section 2.3, the complete IBVP for focused ultrasound applications is presented. In particular, the Westervelt equation in a heterogeneous truncated domain along with the initial conditions are presented. To handle the truncated domain, the formulation of the corresponding absorbing and source boundary condition is discussed. The variational formulation of the IBVP is presented in section 2.4.

It is worth noting here that this chapter is a review of prior works and is included for completeness [74, 93].

2.1 Derivation of the Westervelt equation

To model a focused ultrasound application, the Westervelt equation is used. This model takes into account nonlinear effects and provides a suitable approximation of the full second-order wave equation [74]. Unlike the KZK and the wide angle KZK equation, the Westervelt equation is naturally suited for modelling a strongly focused ultrasound transducer [191]. Additionally, the Westervelt equation is suitable for modelling the nonlinear effect that is important in high-intensity focused ultrasound applications [74, 150, 70, 145, 163].

A brief derivation of the Westervelt equation is presented. A more detailed analysis and derivation can be found in [74]. The propagation of a sound field in a domain Ω is governed by the equation of mass conservation, momentum conservation, entropy balance, and thermodynamic state. The mass conservation equation reads

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \quad (2.1)$$

where $\rho(\mathbf{x}) \in \mathbb{R}^+$ is the medium density and $\mathbf{u}(\mathbf{x}) \in \mathbb{R}^3$ is the fluid velocity vector. The momentum conservation equation reads

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) + \nabla P = \mu \nabla^2 \mathbf{u} + \left(\mu_B + \frac{1}{3} \mu \right) \nabla (\nabla \cdot \mathbf{u}), \quad (2.2)$$

where $P(\mathbf{x}) \in \mathbb{R}$ is the thermodynamic pressure, $\mu(\mathbf{x}) \in \mathbb{R}^+$ is the shear viscosity, and $\mu_B(\mathbf{x}) \in \mathbb{R}^+$ is the bulk viscosity. The entropy balance equation reads

$$\rho T \left(\frac{\partial s}{\partial t} + \mathbf{u} \cdot \nabla s \right) = \kappa \nabla^2 T + \mu_B (\nabla \cdot \mathbf{u})^2 + \frac{1}{2} \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3} \delta_{ij} \frac{\partial u_k}{\partial x_k} \right)^2, \quad (2.3)$$

where $s(\mathbf{x}) \in \mathbb{R}$ is the specific entropy per unit mass, $\kappa(\mathbf{x}) \in \mathbb{R}^+$ is the thermal conductivity, and δ_{ij} is the Kronecker delta. The third term on the right-hand side is written in the Cartesian tensor notation where u_i denotes the component of the velocity vector $\mathbf{u}$ in the x_i direction. The equation of state is given as

$$P = P(\rho, s), \quad (2.4)$$

where the thermodynamic pressure depends on the density and the specific entropy.

The aim is to obtain the second-order approximation of the mass conservation and momentum conservation equation. To this end, consider a perturbation (denoted with primes) of each variables such that

$$\begin{aligned} \rho' &= \rho - \rho_0, \\ p' &= P - P_0, \\ \mathbf{u}' &= \mathbf{u}, \\ s' &= s - s_0, \end{aligned} \quad (2.5)$$

where the subscript '0' denotes an unperturbed state that is constants in space and time.

Substituting equation (2.5) into equation (2.1) yields

$$\frac{\partial \rho'}{\partial t} + \frac{\partial \rho_0}{\partial t} + \nabla \cdot [(\rho' + \rho_0) \mathbf{u}] = 0. \quad (2.6)$$

Since the rate of change with respect to time of the unperturbed state can be eliminated, the equation reduces to

$$\frac{\partial \rho'}{\partial t} + \rho_0 \nabla \cdot \mathbf{u} = -\rho' \nabla \cdot \mathbf{u} - \nabla \rho' \cdot \mathbf{u} \quad (2.7)$$

with the third term on the left-hand side of equation (2.6) expanded. It is remarked here that first-order terms consist of a single perturbed variable and second-order terms consist of a *multiplication* of two first-order terms, e.g., the left-hand side and right-hand side of equation (2.7) consists of first-order and second-order term, respectively. Third and higher-order terms consist of a *multiplication* of more than two perturbed variables.

Substituting equation (2.5) into equation (2.2), gives

$$(\rho' + \rho_0) \left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} \right) + \nabla p' + \nabla P_0 = \mu \nabla^2 \mathbf{u} + \left(\mu_B + \frac{1}{3} \mu \right) \nabla (\nabla \cdot \mathbf{u}). \quad (2.8)$$

Applying the vector identities $(\mathbf{u} \cdot \nabla) \mathbf{u} = \frac{1}{2} \nabla u^2 - \mathbf{u} \times \nabla \times \mathbf{u}$ and $\nabla (\nabla \cdot \mathbf{u}) = \nabla^2 \mathbf{u} + \nabla \times \nabla \times \mathbf{u}$ yield

$$\begin{aligned} \rho_0 \frac{\partial \mathbf{u}}{\partial t} + \nabla p = & \left(\mu_B + \frac{4}{3} \mu \right) \nabla^2 \mathbf{u} - \frac{1}{2} \rho_0 \nabla u^2 - \rho' \frac{\partial \mathbf{u}}{\partial t} + \left(\mu_B + \frac{1}{3} \mu \right) (\nabla \times \nabla \times \mathbf{u}) \\ & + \rho_0 (\mathbf{u} \times \nabla \times \mathbf{u}) - \frac{1}{2} \rho' \nabla u^2 + \rho' (\mathbf{u} \times \nabla \times \mathbf{u}), \end{aligned} \quad (2.9)$$

where $u = \sqrt{\mathbf{u} \cdot \mathbf{u}}$. Eliminating third-order terms (terms in blue), yields

$$\begin{aligned} \rho_0 \frac{\partial \mathbf{u}}{\partial t} + \nabla p = & \left(\mu_B + \frac{4}{3} \mu \right) \nabla^2 \mathbf{u} - \frac{1}{2} \rho_0 \nabla u^2 - \rho' \frac{\partial \mathbf{u}}{\partial t} + \left(\mu_B + \frac{1}{3} \mu \right) (\nabla \times \nabla \times \mathbf{u}) \\ & + \rho_0 (\mathbf{u} \times \nabla \times \mathbf{u}). \end{aligned} \quad (2.10)$$

Assuming irrotational $\mathbf{u}$ (valid for focused ultrasound model [74]), i.e. $\nabla \times \mathbf{u} = 0$, gives

$$\rho_0 \frac{\partial \mathbf{u}}{\partial t} + \nabla p = \left(\mu_B + \frac{4}{3} \mu \right) \nabla^2 \mathbf{u} - \frac{1}{2} \rho_0 \nabla u^2 - \rho' \frac{\partial \mathbf{u}}{\partial t}. \quad (2.11)$$

To obtain the second-order approximation of the entropy balance equation, equation (2.5) is substituted into equation (2.3), which gives

$$\rho_0 T_0 \frac{\partial s'}{\partial t} = \kappa \nabla^2 T'. \quad (2.12)$$

Finally, the equation of state in equation (2.4) is expanded using a Taylor series about the equilibrium state (ρ_0, s_0) and the second-order terms are retained, to yield

$$p = c_0^2 \rho' + \frac{c_0^2}{\rho_0} \frac{B}{2A} \rho'^2 + \left(\frac{\partial P}{\partial s} \right)_{\rho,0} s', \quad (2.13)$$

where $B/A = (\rho_0/c_0^2)(\partial^2 P/\partial \rho^2)_{s,0}$ is the nonlinearity parameter. This set of equations is the basis of the nonlinear Westervelt equation. The k-Wave solver solves this specific set of equations [160]. Equations (2.7) and (2.11) to (2.13) are the second-order approximation of the conservation equations and equation of state. The k-Wave solver is able to handle arbitrary frequency attenuation power

laws, which uses two lossy derivative operators that represent the required power law absorption and dispersion [161]. However, for nonlinear problems, it is limited to a single attenuation power law applied uniformly across the entire domain.

To obtain a single wave equation, the second-order terms are substituted with the first-order approximation [74]. The first-order approximation of the mass conservation, momentum conservation and equation of state are

$$\frac{\partial \rho'}{\partial t} + \rho_0 \nabla \cdot \mathbf{u} = 0, \quad (2.14a)$$

$$\rho_0 \frac{\partial \mathbf{u}}{\partial t} + \nabla p = 0, \quad (2.14b)$$

$$p = c_0^2 \rho'. \quad (2.14c)$$

Substituting equation (2.14) into equation (2.7) yields

$$\frac{\partial \rho'}{\partial t} + \rho_0 \nabla \cdot \mathbf{u} = \frac{1}{\rho_0 c_0^4} \frac{\partial p^2}{\partial t} + \frac{1}{c_0^2} \frac{\partial L}{\partial t}, \quad (2.15)$$

where $L = \rho_0 u^2 / 2 - p^2 / (2\rho_0 c_0^2)$ is the second-order Lagrangian density [74] that represents the difference between the kinetic and potential energy. Similarly, substituting equation (2.14) into equation (2.11) yields

$$\rho_0 \frac{\partial \mathbf{u}}{\partial t} + \nabla p = -\frac{\mu_B + \frac{4}{3}\mu}{\rho_0 c_0^2} \nabla \frac{\partial p}{\partial t} - \nabla L. \quad (2.16)$$

Applying the linear relation $\nabla^2 T' = c_0^{-2} \partial^2 T' / \partial t^2$, equation (2.12) can be rewritten as

$$\frac{\rho_0 T_0}{\kappa} \frac{\partial s'}{\partial t} = \frac{1}{c_0^2} \frac{\partial^2 T'}{\partial t^2} \Rightarrow s' = \frac{\kappa}{\rho_0 c_0^2 T_0} \frac{\partial T'}{\partial t}. \quad (2.17)$$

Substituting equations (2.14c) and (2.17) into equation (2.13), which yields

$$\rho' = \frac{1}{c_0^2} p - \frac{1}{\rho_0 c_0^4} \frac{B}{2A} p^2 - \frac{\kappa}{\rho_0 c_0^4 T_0} \left(\frac{\partial p}{\partial s} \right)_{\rho,0} \left(\frac{\partial T'}{\partial t} \right). \quad (2.18)$$

Eliminating T' by expanding $T = T(\rho, s)$ using a truncated Taylor expansion around ρ_0 such that

$$T = T_0 + \left(\frac{\partial T}{\partial \rho} \right)_{s,0} (\rho - \rho_0) + \dots \quad (2.19)$$

Rearranging equation (2.19) and retaining only the first-order term gives

$$T - T_0 = \left(\frac{\partial T}{\partial \rho} \right)_{s,0} (\rho - \rho_0) \Rightarrow T' = \left(\frac{\partial T}{\partial \rho} \right)_{s,0} \rho'. \quad (2.20)$$

Substituting equations (2.14c) and (2.20) into equation (2.18) yields

$$\rho' = \frac{1}{c_0^2} p - \frac{1}{\rho_0 c_0^4} \frac{B}{2A} p^2 - \frac{\kappa}{\rho_0 c_0^6 T_0} \left(\frac{\partial P}{\partial s} \right)_{\rho,0} \left(\frac{\partial T}{\partial \rho} \right)_{s,0} \frac{\partial p}{\partial t}. \quad (2.21)$$

Applying the thermodynamic relations $(\partial P / \partial s)_\rho = \rho^2 (\partial T / \partial \rho)_s$ and $(\partial T / \partial \rho)_s^2 = R T c^2 / (c_v c_p \rho^2)$ where $R = c_p - c_v$, and after some algebraic manipulation, equation (2.21) can be written as

$$\rho' = \frac{1}{c_0^2} p - \frac{1}{\rho_0 c_0^4} \frac{B}{2A} p^2 - \frac{\kappa}{\rho_0 c_0^4} \left(\frac{1}{c_v} - \frac{1}{c_p} \right) \frac{\partial p}{\partial t}, \quad (2.22)$$

where R is the gas constant, c_v is the specific-heat coefficient at constant pressure and c_p is the specific-heat coefficient at constant volume. To retain only the pressure variable, $\mathbf{u}$ is eliminated by subtracting the derivative with respect to time of equation (2.15) from the divergence of equation (2.16), which reduces to

$$\frac{1}{\rho_0} \nabla^2 p - \frac{1}{\rho_0} \frac{\partial^2 \rho'}{\partial t^2} = -\frac{\mu_B + \frac{4}{3}\mu}{\rho_0^2 c_0^2} \nabla^2 \frac{\partial p}{\partial t} - \frac{1}{\rho_0 c_0^4} \frac{\partial^2 p^2}{\partial t^2} - \frac{1}{\rho_0} \left[\nabla^2 + \frac{1}{c_0^2} \frac{\partial^2}{\partial t^2} \right] L. \quad (2.23)$$

Finally, ρ' is eliminated by substituting equation (2.22) into equation (2.23) which gives

$$\frac{1}{\rho_0} \nabla^2 p - \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} + \frac{\delta}{\rho_0 c_0^2} \nabla^2 \frac{\partial p}{\partial t} = -\frac{\beta}{\rho_0^2 c_0^4} \frac{\partial^2 p^2}{\partial t^2} - \frac{1}{\rho_0} \left[\nabla^2 + \frac{1}{c_0^2} \frac{\partial^2}{\partial t^2} \right] L \quad (2.24)$$

where

$$\delta = \frac{\mu_B + \frac{4}{3}\mu}{\rho_0} + \frac{\kappa}{\rho_0} \left(\frac{1}{c_v} - \frac{1}{c_p} \right) \quad (2.25)$$

is the diffusivity of sound and

$$\beta = 1 + \frac{B}{2A} \quad (2.26)$$

is the nonlinearity coefficient. In the case where the cumulative nonlinear effects dominate the local nonlinear effect, which is the case for focused ultrasound applications, the Lagrangian density can be ignored [74]. Thus, the Westervelt equation is obtained, which reads

$$\frac{1}{\rho_0} \nabla^2 p - \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} + \frac{\delta}{\rho_0 c_0^2} \nabla^2 \frac{\partial p}{\partial t} = -\frac{\beta}{\rho_0^2 c_0^4} \frac{\partial^2 p^2}{\partial t^2}. \quad (2.27)$$

It is worth noting here that the diffusivity term of equation (2.27) assumes a quadratic dependence of the source frequency [162, 95]. In the rest of the thesis, the focused is on the solution of equation (2.27).

2.2 Sommerfeld radiation condition

In this section, a short description of the Sommerfeld radiation condition that is important for exterior boundary value problems is presented. An exterior boundary value problem is a boundary value problem where the boundary condition must include a statement of the solution behaviour at infinity. That statement is the radiation condition.

The statement of the Sommerfeld radiation condition is [146]

$$\frac{\partial u}{\partial r} + \frac{1}{c_0} \frac{\partial u}{\partial t} = o\left(\frac{1}{r^{(d-1)/2}}\right) \quad (2.28)$$

as $r := |\mathbf{x}| \rightarrow \infty$, uniformly in $\mathbf{x}/r$. A solution that satisfies the Sommerfeld radiation condition is called a radiating solution. The condition ensures that a scattered wave is only outgoing at infinity, i.e., there are no reflected waves at infinity. The radiation condition is necessary to ensure that the solution is unique and *physically* valid [139].

2.3 Statement of the initial-boundary value problem

The initial-value boundary problem is formulated using the Westervelt equation of equation (2.27) and the Sommerfeld radiation condition of equation (2.28). The initial-boundary value problem reads: given $\rho_0(\mathbf{x})$, $c_0(\mathbf{x})$, $\delta(\mathbf{x})$, $\beta(\mathbf{x}) \in \mathbb{R}^+$, find p such that

$$\frac{1}{\rho_0} \nabla^2 p - \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} + \frac{\delta}{\rho_0 c_0^2} \nabla^2 \frac{\partial p}{\partial t} = -\frac{\beta}{\rho_0^2 c_0^4} \frac{\partial^2 p^2}{\partial t^2} \quad \text{in } \Omega \times (0, T), \quad (2.29a)$$

$$p|_{t=0} = 0, \quad \left. \frac{\partial p}{\partial t} \right|_{t=0} = 0 \quad \text{in } \Omega, \quad (2.29b)$$

with p satisfying the Sommerfeld radiation condition such that

$$\frac{\partial p}{\partial r} + \frac{1}{c_0} \frac{\partial p}{\partial t} = o\left(\frac{1}{r^{(d-1)/2}}\right) \quad (2.29c)$$

as $r := |\mathbf{x}| \rightarrow \infty$, uniformly in $\mathbf{x}/r$. In the above, the superscript d is the geometric dimension and T is the final time. Equation (2.29b) is the initial condition of the initial-boundary value problem. In order to solve equation (2.29) using the finite element method, the unbounded domain problem is truncated into a bounded domain problem while ensuring that the solution of the bounded domain problem is the same as the unbounded domain problem. In practice, this is impossible to achieve since there will always be some reflections at the boundary of the truncated domain problem. However, for many engineering applications, there are techniques such as absorbing boundary conditions (ABC) and perfectly matched layer (PML) that can reduce the reflections to an acceptable level.

A widely used technique is the absorbing boundary conditions which seek to approximate equation (2.28). In this regard, Engquist and Majda presented several approximations to equation (2.28) in

[53]. They show 1st, 2nd and 3rd order approximations to the radiation condition. There exist other method to absorb outgoing waves, such as the perfectly matched layer [21, 92, 127] and adiabatically absorbing layer [120, 152]. Even though these method are more accurate, they are also computationally more expensive. As such, the absorbing boundary conditions approach is employed due to its computational efficiency and ease of implementation.

Consider the first-order ABC. This is equivalent to imposing the Sommerfeld radiation condition on the boundary of the truncated domain instead of as $r \rightarrow \infty$. More precisely

$$\frac{\partial p}{\partial \mathbf{n}} + \frac{1}{c_0} \frac{\partial p}{\partial t} = 0 \quad \text{at } \Gamma, \quad (2.30)$$

where Γ is the boundary of the computational domain Ω and $\mathbf{n}$ is the normal vector at the domain boundary. Before proceeding the modified boundary value problem, an excitation function needs to be imposed, i.e., the source function. To this end, consider a function that is defined such that

$$\frac{\partial p}{\partial \mathbf{n}} + \frac{1}{c_0} \frac{\partial p}{\partial t} = g(t) \quad \text{on } \Gamma_s, \quad (2.31)$$

where $g(t)$ is a time-dependent function that is imposed at a specific region on the domain boundary Γ_s . This approach is equivalent to imposing a Neumann boundary condition on the domain boundary [132]. It should be noted that another widely used approach to model the transducer source is by using a collection of monopole point sources [127, 160, 75, 43].

The modified initial-boundary value problem reads: given $\rho_0(\mathbf{x})$, $c_0(\mathbf{x})$, $\delta(\mathbf{x})$, $\beta(\mathbf{x}) \in \mathbb{R}^+$, find p such that

$$\frac{1}{\rho_0} \nabla^2 p - \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} + \frac{\delta}{\rho_0 c_0^2} \nabla^2 \frac{\partial p}{\partial t} = -\frac{\beta}{\rho_0^2 c_0^4} \frac{\partial^2 p^2}{\partial t^2} \quad \text{in } \Omega \times (0, T), \quad (2.32a)$$

$$p|_{t=0} = 0, \quad \left. \frac{\partial p}{\partial t} \right|_{t=0} = 0 \quad \text{in } \Omega, \quad (2.32b)$$

with boundary conditions such that

$$\frac{\partial p}{\partial \mathbf{n}} + \frac{1}{c_0} \frac{\partial p}{\partial t} = 0 \quad \text{at } \Gamma_a \quad (2.32c)$$

and

$$\frac{\partial p}{\partial \mathbf{n}} + \frac{1}{c_0} \frac{\partial p}{\partial t} = g(t) \quad \text{at } \Gamma_s, \quad (2.32d)$$

where $\Gamma_a \cup \Gamma_s = \Gamma$.

2.4 Variational formulation of the initial-boundary value problem

In order for equation (2.32) to be solved using the finite element method, the IBVP needs to be cast into a variational form. Following [93], equation (2.32a) is multiplied with the test function w and

integrating over the domain Ω which gives

$$\int_{\Omega} \frac{1}{\rho_0} (\nabla^2 p) w \, d\Omega - \int_{\Omega} \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} w \, d\Omega + \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \left(\nabla^2 \frac{\partial p}{\partial t} \right) w \, d\Omega = - \int_{\Omega} \frac{\beta}{\rho_0^2 c_0^4} \frac{\partial^2 p^2}{\partial t^2} w \, d\Omega. \quad (2.33)$$

Applying integration by parts to the first and third terms on the left-hand side gives

$$\begin{aligned} & - \int_{\Omega} \frac{1}{\rho_0} \nabla p \cdot \nabla w \, d\Omega + \int_{\Gamma} \frac{1}{\rho_0} \frac{\partial p}{\partial \mathbf{n}} w \, d\Gamma - \int_{\Omega} \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} w \, d\Omega - \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \nabla \frac{\partial p}{\partial t} \cdot \nabla w \, d\Omega \\ & + \int_{\Gamma} \frac{\delta}{\rho_0 c_0^2} \frac{\partial}{\partial t} \left(\frac{\partial p}{\partial \mathbf{n}} w \right) \, d\Gamma = - \int_{\Omega} \frac{\beta}{\rho_0^2 c_0^4} \frac{\partial^2 p^2}{\partial t^2} w \, d\Omega. \end{aligned} \quad (2.34)$$

Applying the boundary condition from equations (2.32c) and (2.32d) yields

$$\begin{aligned} & - \int_{\Omega} \frac{1}{\rho_0} \nabla p \cdot \nabla w \, d\Omega - \int_{\Gamma} \frac{1}{\rho_0 c_0} \frac{\partial p}{\partial t} w \, d\Gamma + \int_{\Gamma_s} \frac{1}{\rho_0} g w \, d\Gamma_s - \int_{\Omega} \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} w \, d\Omega \\ & - \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \nabla \frac{\partial p}{\partial t} \cdot \nabla w \, d\Omega - \int_{\Gamma} \frac{\delta}{\rho_0 c_0^3} \frac{\partial^2 p}{\partial t^2} w \, d\Gamma + \int_{\Gamma_s} \frac{\delta}{\rho_0 c_0^2} \frac{\partial g}{\partial t} w \, d\Gamma_s = - \int_{\Omega} \frac{\beta}{\rho_0^2 c_0^4} \frac{\partial^2 p^2}{\partial t^2} w \, d\Omega, \end{aligned} \quad (2.35)$$

where $\Gamma = \Gamma_a \cup \Gamma_s$. Expanding the term on the right-hand side using the product rule yields

$$\begin{aligned} & - \int_{\Omega} \frac{1}{\rho_0} \nabla p \cdot \nabla w \, d\Omega - \int_{\Gamma} \frac{1}{\rho_0 c_0} \frac{\partial p}{\partial t} w \, d\Gamma + \int_{\Gamma_s} \frac{1}{\rho_0} g w \, d\Gamma_s - \int_{\Omega} \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} w \, d\Omega \\ & - \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \nabla \frac{\partial p}{\partial t} \cdot \nabla w \, d\Omega - \int_{\Gamma} \frac{\delta}{\rho_0 c_0^3} \frac{\partial^2 p}{\partial t^2} w \, d\Gamma + \int_{\Gamma_s} \frac{\delta}{\rho_0 c_0^2} \frac{\partial g}{\partial t} w \, d\Gamma_s \\ & = - \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} p \frac{\partial^2 p}{\partial t^2} w \, d\Omega - \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} \left(\frac{\partial p}{\partial t} \right)^2 w \, d\Omega. \end{aligned} \quad (2.36)$$

Following [143], the space H^1 where $p \in L^2(0, T; H^1(\Omega))$, $\partial p / \partial t \in L^2(0, T; L^2(\Omega))$, $\partial^2 p / \partial t^2 \in L^2(0, T; H^{-1}(\Omega))$, and $w \in H^1$ at all times $t \in (0, T)$ is used. Accordingly, the variational problem of equation (2.32) reads: find $p \in L^2(0, T; H^1(\Omega))$ such that

$$\begin{aligned} & - \int_{\Omega} \frac{1}{\rho_0} \nabla p \cdot \nabla w \, d\Omega - \int_{\Gamma} \frac{1}{\rho_0 c_0} \frac{\partial p}{\partial t} w \, d\Gamma + \int_{\Gamma_s} \frac{1}{\rho_0} g w \, d\Gamma_s - \int_{\Omega} \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} w \, d\Omega \\ & - \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \nabla \frac{\partial p}{\partial t} \cdot \nabla w \, d\Omega - \int_{\Gamma} \frac{\delta}{\rho_0 c_0^3} \frac{\partial^2 p}{\partial t^2} w \, d\Gamma + \int_{\Gamma_s} \frac{\delta}{\rho_0 c_0^2} \frac{\partial g}{\partial t} w \, d\Gamma_s \\ & = - \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} p \frac{\partial^2 p}{\partial t^2} w \, d\Omega - \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} \left(\frac{\partial p}{\partial t} \right)^2 w \, d\Omega \end{aligned} \quad (2.37)$$

for all $w \in H^1$ at all times $t \in (0, T)$. The variational problem above has been proven to be well-posed [93]. In the rest of this thesis, the finite element solution of equation (2.37) is of interest.

2.5 Summary

In this chapter, the derivation of a PDE that models a focused ultrasound application, namely the Westervelt equation, was presented. The Sommerfeld radiation condition, which is important for exterior boundary value problems, was discussed. The combination of the Westervelt equation, the Sommerfeld radiation condition, and the initial conditions defines the initial-boundary value problem for focused ultrasound applications.

Chapter 3

Numerical formulation of a focused ultrasound problem

Due to the complexity of a focused ultrasound application, the continuous IBVP presented in chapter 2 cannot be solved analytically. This motivates the development of numerical methods. In this chapter, the continuous variational IBVP of equation (2.37) is formulated into a fully discrete system that is suited for computer implementation.

In section 3.1, the continuous problem of equation (2.37) is converted into a semi-discrete form using a Galerkin approximation. Next, in section 3.2, the fully discrete form is formulated using the explicit Runge–Kutta method and the 4th order explicit Runge–Kutta method employed in this thesis is briefly discussed. Section 3.3 presents a specific type of finite element scheme, namely the nodal quadrature mass-lumped finite element scheme. In particular, the construction and the justification for the mass lumped finite element scheme is presented. In section 3.4, the degree-of-freedom map that enables the implementation of arbitrary order polynomial basis functions on quadrilateral and hexahedral meshes is discussed. In section 3.5, an overview of the solver algorithm is provided.

3.1 Galerkin approximation of the variational form

Given the continuous IBVP of equation (2.32), the Galerkin approximation converts the continuous problem into a semi-discrete problem by restricting an approximate solution to a finite-dimensional subspace, $S_h \subset H^1(\Omega)$. In particular, instead of seeking a solution in $H^1(\Omega)$, an approximate solution in S_h is sought, which makes the problem computable. Accordingly, equation (2.32) is reformulated such that: let $S_h \subset H^1(\Omega)$, the Galerkin approximation of equation (2.32) reads: given $g(t) \in$

$L^2(0, T; S_h(\Omega))$ and $\rho_0(\mathbf{x}), c_0(\mathbf{x}), \delta(\mathbf{x}), \beta(\mathbf{x}) \in \mathbb{R}^+$, find $p_h \in L^2(0, T; S_h(\Omega))$ such that

$$\begin{aligned} & - \int_{\Omega} \frac{1}{\rho_0} \nabla p_h \cdot \nabla w_h \, d\Omega - \int_{\Gamma} \frac{1}{\rho_0 c_0} \frac{\partial p_h}{\partial t} w_h \, d\Gamma + \int_{\Gamma_s} \frac{1}{\rho_0} g w_h \, d\Gamma_s - \int_{\Omega} \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p_h}{\partial t^2} w_h \, d\Omega \\ & - \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \nabla \frac{\partial p_h}{\partial t} \cdot \nabla w_h \, d\Omega - \int_{\Gamma} \frac{\delta}{\rho_0 c_0^3} \frac{\partial^2 p_h}{\partial t^2} w_h \, d\Gamma + \int_{\Gamma_s} \frac{\delta}{\rho_0 c_0^2} \frac{\partial g}{\partial t} w_h \, d\Gamma_s \\ & = - \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} p_h \frac{\partial^2 p_h}{\partial t^2} w_h \, d\Omega - \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} \left(\frac{\partial p_h}{\partial t} \right)^2 w_h \, d\Omega \quad (3.1) \end{aligned}$$

for all $w_h \in S_h$ at all times $t \in (0, T)$. The finite dimensional subspace S_h is spanned by the set of basis functions

$$S_h = \text{span} \{ \phi_1(x), \phi_2(x), \dots, \phi_n(x) \}, \quad (3.2)$$

such that every function in S_h can be expressed as

$$p_h(x) = \sum_{i=1}^n p_i \phi_i(x), \quad (3.3)$$

where p_i are the degrees-of-freedom. To solve equation (3.1) using a computer, the semi-discrete equation needs to be converted into a fully discrete equation.

3.2 Fully discrete formulation using the explicit Runge–Kutta method

The Runge–Kutta method is a numerical method for solving ordinary differential equation (ODE) [30, 72, 177]. Given an ODE

$$\frac{dy}{dt} = f(t), \quad (3.4)$$

an s -stage Runge–Kutta method can be written as

$$y^{n+1} = y^n + \Delta t \sum_{i=1}^s b_i k_i, \quad (3.5)$$

where

$$k_i = f(t^n + c_i \Delta t, y^n + \Delta t \sum_{j=1}^s a_{ij} k_j), \quad i = 1, \dots, s \quad (3.6)$$

and Δt is the temporal step-size, and the constants a_{ij} , b_i and $c_i = \sum_j a_{ij}$ are the coefficients of the method. These coefficients determines the type and order of the Runge–Kutta method. Typically, the coefficients are presented in a Butcher-tableau.

To apply the Runge–Kutta method and obtain a fully discrete form of equation (3.1), equation (3.1) is split into a system of first-order ordinary differential equations (ODE). To this end, the time-

derivative of the pressure field p_h is introduced such that

$$v_h = \frac{\partial p_h}{\partial t} \Rightarrow \frac{\partial v_h}{\partial t} = \frac{\partial^2 p_h}{\partial t^2}. \quad (3.7)$$

Substituting equation (3.7) into equation (3.1) and rearranging yields

$$\begin{aligned} \int_{\Omega} \left(\frac{1}{\rho_0 c_0^2} - \frac{2\beta}{\rho_0^2 c_0^4} p_h \right) \frac{\partial v_h}{\partial t} w_h \, d\Omega + \int_{\Gamma} \frac{\delta}{\rho_0 c_0^3} \frac{\partial v_h}{\partial t} w_h \, d\Gamma + \int_{\Omega} \frac{1}{\rho_0} \nabla p_h \cdot \nabla w_h \, d\Omega \\ + \int_{\Gamma} \frac{1}{\rho_0 c_0} v_h w_h \, d\Gamma + \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \nabla v_h \cdot \nabla w_h \, d\Omega - \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} v_h^2 w_h \, d\Omega \\ = \int_{\Gamma_s} \left(\frac{1}{\rho_0} g + \frac{\delta}{\rho_0 c_0^2} \frac{\partial g}{\partial t} \right) w_h \, d\Gamma_s. \end{aligned} \quad (3.8)$$

Accordingly, the system of two *first-order* ODEs in $\partial p_h / \partial t$ and $\partial v_h / \partial t$ is given as

$$\int_{\Omega} \frac{\partial p_h}{\partial t} w_h \, d\Omega = \int_{\Omega} v_h w_h \, d\Omega, \quad (3.9a)$$

$$\begin{aligned} \int_{\Omega} \left(\frac{1}{\rho_0 c_0^2} - \frac{2\beta}{\rho_0^2 c_0^4} p_h \right) \frac{\partial v_h}{\partial t} w_h \, d\Omega + \int_{\Gamma} \frac{\delta}{\rho_0 c_0^3} \frac{\partial v_h}{\partial t} w_h \, d\Gamma = - \int_{\Omega} \frac{1}{\rho_0} \nabla p_h \cdot \nabla w_h \, d\Omega \\ - \int_{\Gamma} \frac{1}{\rho_0 c_0} v_h w_h \, d\Gamma - \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \nabla v_h \cdot \nabla w_h \, d\Omega + \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} v_h^2 w_h \, d\Omega \\ + \int_{\Gamma_s} \left(\frac{1}{\rho_0} g_h + \frac{\delta}{\rho_0 c_0^2} \frac{\partial g_h}{\partial t} \right) w_h \, d\Gamma_s. \end{aligned} \quad (3.9b)$$

Using an explicit Runge–Kutta method, equation (3.9) can be expressed as a linear system such that

$$\mathbf{M}\dot{\mathbf{p}} = \mathbf{M}\mathbf{v}, \quad (3.10a)$$

$$(\mathbf{M}(\mathbf{p}) + \mathbf{M}_{\Gamma})\dot{\mathbf{v}} = \mathbf{f}(\mathbf{p}) + \mathbf{f}(\mathbf{v}) + \mathbf{g}, \quad (3.10b)$$

where $\dot{\mathbf{p}}$ and $\dot{\mathbf{v}}$ are the unknown vectors. The matrices $\mathbf{M}$, $\mathbf{M}(\mathbf{p})$, and $\mathbf{M}_{\Gamma}$ are the mass matrix where

$$\mathbf{M} = \int_{\Omega} \frac{\partial p_h}{\partial t} w_h \, d\Omega, \quad (3.11a)$$

$$\mathbf{M}(\mathbf{p}) = \int_{\Omega} \left(\frac{1}{\rho_0 c_0^2} - \frac{2\beta}{\rho_0^2 c_0^4} p_h \right) \frac{\partial v_h}{\partial t} w_h \, d\Omega, \quad (3.11b)$$

$$\mathbf{M}_{\Gamma} = \int_{\Gamma} \frac{\delta}{\rho_0 c_0^3} \frac{\partial v_h}{\partial t} w_h \, d\Gamma. \quad (3.11c)$$

These mass matrices differ in terms of their integration domain and integrand. The vectors $\mathbf{f}(\mathbf{p})$, $\mathbf{f}(\mathbf{v})$, and $\mathbf{g}$ are the known vectors where

$$\mathbf{f}(\mathbf{p}) = - \int_{\Omega} \frac{1}{\rho_0} \nabla p_h \cdot \nabla w_h \, d\Omega, \quad (3.12a)$$

$$\mathbf{f}(\mathbf{v}) = - \int_{\Gamma} \frac{1}{\rho_0 c_0} v_h w_h \, d\Gamma - \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \nabla v_h \cdot \nabla w_h \, d\Omega + \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} v_h^2 w_h \, d\Omega, \quad (3.12b)$$

$$\mathbf{g} = \int_{\Gamma_s} \left(\frac{1}{\rho_0} g_h + \frac{\delta}{\rho_0 c_0^2} \frac{\partial g_h}{\partial t} \right) w_h \, d\Gamma_s. \quad (3.12c)$$

Given a system of *first-order* ODEs of equation (3.10), an s -stage Runge–Kutta method can be written as

$$\mathbf{p}^{n+1} = \mathbf{p}^n + \Delta t \sum_{i=1}^s b_i \mathbf{k}_i^p, \quad (3.13)$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \Delta t \sum_{i=1}^s b_i \mathbf{k}_i^v, \quad (3.14)$$

where

$$\begin{aligned} \mathbf{k}_i^p &= f^p(t^n + c_i \Delta t, \mathbf{p}^n + \Delta t \sum_{j=1}^s a_{ij} \mathbf{k}_j^p, \mathbf{v}^n + \Delta t \sum_{j=1}^s a_{ij} \mathbf{k}_j^v) \\ &= \dot{\mathbf{p}} \\ &= \mathbf{M}^{-1} \mathbf{M} \mathbf{v}_i \\ &= \mathbf{v}_i, \end{aligned} \quad (3.15a)$$

$$\begin{aligned} \mathbf{k}_i^v &= f^v(t^n + c_i \Delta t, \mathbf{p}^n + \Delta t \sum_{j=1}^s a_{ij} \mathbf{k}_j^p, \mathbf{v}^n + \Delta t \sum_{j=1}^s a_{ij} \mathbf{k}_j^v) \\ &= \dot{\mathbf{v}} \\ &= (\mathbf{M}(\mathbf{p}_i) + \mathbf{M}_{\Gamma})^{-1} (\mathbf{f}(\mathbf{p}_i) + \mathbf{f}(\mathbf{v}_i) + \mathbf{g}_i), \end{aligned} \quad (3.15b)$$

for $i = 1, \dots, s$. In this thesis, the classical 4th order explicit Runge–Kutta scheme is employed and the Butcher tableau is shown in table 3.1. A benefit of the 4th order explicit Runge–Kutta scheme is that the computational work is low since its tableau contains only diagonal entries, and as such the intermediate time step depends entirely on the previous time step. The scheme has low storage requirements [181, 96, 98]. At each Runge–Kutta time step only a single vector needs to be stored for $\mathbf{p}_i$ and $\mathbf{v}_i$.

Table 3.1 Butcher-tableau for the classical 4th order explicit Runge–Kutta scheme.

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

3.3 Mass-lumped finite element method

The Galerkin approximation in section 3.1 is extremely general. In fact, the subspace S_h could be chosen from several spaces, such as wavelets, spectral expansion, boundary element, or finite element [54]. In this work, the finite element method is utilised. The method is naturally parallelisable, which is advantageous for 3D simulations. Most importantly, in conjunction with an explicit Runge–Kutta method as presented in section 3.2, a mass-lumped finite element scheme avoids the need for solving a linear system, hence reduces the computational cost of the solver.

Equation (3.15) is discretised using the finite element method. Piecewise polynomial basis functions are considered for the construction of S_h [54]. The domain Ω is partitioned into N_c disjoint elements K , where K is a quadrilateral in 2D and hexahedron in 3D, which together comprise the mesh such that

$$\overline{\Omega} = \cup_i K_i \quad i = 1, \dots, N_c. \quad (3.16)$$

On a given mesh of Ω and for a given approximation degree $l \geq 1$, the finite dimensional subspace S_h is defined as

$$S_h = \{w \in H^1(\Omega) : w|_K \in Q_l(K), \forall K\}, \quad l \geq 1, \quad (3.17)$$

where $Q_l(K)$ is the space of piecewise polynomials of maximal degree l [65]. Piecewise Lagrange basis functions are employed, whereby given the points $x_i, i = 0, \dots, N-1$, the associated basis satisfies

$$\phi_i(x_j) = \delta_{ij}, \quad (3.18)$$

where δ_{ij} is the Kronecker delta (not to be confused with the diffusivity of sound). This definition is attractive since it provides the flexibility to define the interpolation points, which gives different variants of the Lagrange element. Customarily, the points of the degrees-of-freedom for quadrilateral and hexahedral finite elements are chosen to be equispaced. However, this choice leads to poorly behaved basis functions—exhibiting Runge’s phenomenon—which is especially pronounced at high-order [140]. A better choice of interpolation points for high-order Lagrange basis is the Gauss–Lobatto–Legendre (GLL) points. The GLL points are the roots of the derivative of the Legendre polynomial, which is defined as

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n. \quad (3.19)$$

Figures 3.1 and 3.2 show the location of the GLL points for quadrilateral and hexahedral elements, respectively. Note that the location of the degrees-of-freedom are the tensor-product of the 1D GLL points.

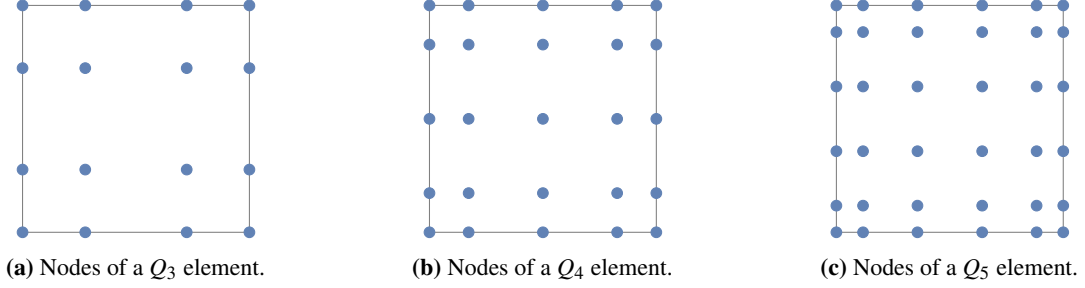


Fig. 3.1 Examples of the Lagrange Q_l finite element on a quadrilateral with nodes at Gauss-Lobatto-Legendre (GLL) points.

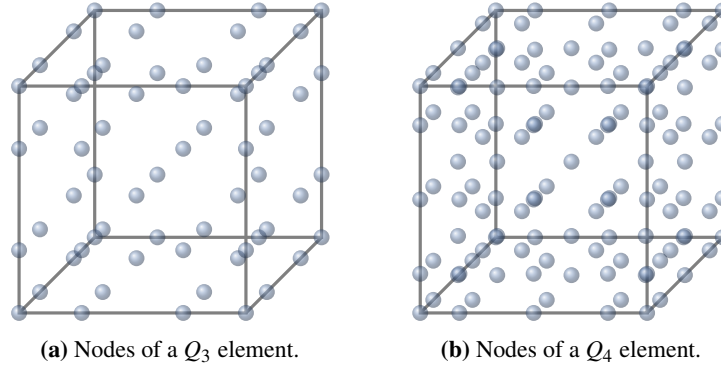


Fig. 3.2 Examples of the Lagrange Q_l finite element on a hexahedron with nodes at Gauss-Lobatto-Legendre (GLL) points.

While several techniques exist for a mass-lumped scheme [52], this thesis uses the nodal quadrature method due to its straightforward implementation [14, 140, 20]. In this method, in conjunction with the GLL basis, the evaluation of integrals, such as in equation (3.15b), is performed using GLL quadrature. If the quadrature points are chosen to coincide with the interpolation points of the GLL basis, the resulting mass matrix is diagonal, a consequence of condition in equation (3.18). To illustrate, consider the evaluation of the entries of the mass matrix on the reference element—an element that provides a convenient way to compute the integral since the formula for the basis functions is the same for all elements and is independent of the element geometry. Additionally, the integration domain is similar for all elements [106]. On the reference element, using GLL quadrature with weights α_q at N_q quadrature points x_q , the entries of the mass matrix is given as

$$M_{ij}^K = \sum_{q=1}^{N_q} \alpha_q \hat{\phi}_i(\hat{x}_q) \hat{\phi}_j(\hat{x}_q) |J_K| \quad (3.20)$$

where the hat symbol denotes the basis function on the reference element and $|J_K|$ denotes the determinant of the Jacobian matrix—a matrix that relates the transformation between the physical and reference element. If the GLL quadrature points are used as basis interpolation points, equation (3.20) reduces to

$$M_{ij}^K = \alpha_q |J_K| \delta_{ij} \Rightarrow M_{ii}^K = \alpha_q |J_K| \delta_{ii} = \alpha_q |J_K|, \quad (3.21)$$

which verifies that the mass matrix is a diagonal matrix. Additionally, from equation (3.21), it is observed that not only is the matrix diagonal, it involves only the multiplication of the Jacobian determinant with the weights of the GLL quadrature. As a result, reduces the number of computations in the assembly of the local mass matrix—a feature that is exploited in chapter 5.

Overall, a diagonal mass matrix provides several computational benefits. Firstly, if use in conjunction with an explicit Runge–Kutta method, a linear solve at each time step can be avoided. Instead, only a vector scaling process is required at each time step. Secondly, since only the diagonal entries are nonzero, the diagonal matrix can be stored as a vector instead of as a matrix and as such reduces the memory requirements of the implementation.

3.4 Degree-of-freedom map for high-order basis

A key aspect of the solver is the use of high-order polynomial basis functions. An advantage of using high-order basis is that it significantly reduces pollution error [84, 18, 85]. In addition, by using high-order basis, the number of element per wavelength required to fully resolve the problem can be reduced [84]. However, to implement a solver that can handle arbitrary order basis function on quadrilateral or hexahedral cells is not trivial, since it requires a proper handling of the mapping between the reference and physical element, especially for degrees-of-freedom on the element's edges and faces. Thus, an implementation that can ensure the correct degree-of-freedom transformations is necessary.

In this section, by way of illustrative example, a brief discussion of the degree-of-freedom map related to the solver implementation is presented, in particular, the degree-of-freedom map for arbitrary order continuous Lagrange finite element. A more general and detail treatment of the degree-of-freedom map can be found in [141].

Since high-degree polynomial Lagrange spaces are of interest, consider the degree 3 Lagrange element defined over the reference quadrilateral as shown in figure 3.3. Each circle with the index i represents the reference degree-of-freedom $\hat{l}_i \in \hat{L}$ where $\hat{L}$ is the basis of the dual space defined based on Ciarlet [35]. The Ciarlet definition defines a finite element by a triple (R, V, L) , where

- $R \subset \mathbb{R}^d$ is the reference element, usually a polygon or a polyhedron.
- V is a finite dimensional polynomial space on R of dimension n .
- $L = \{l_0, \dots, l_{n-1}\}$ is a basis of the dual space $V^* = \{f : V \rightarrow \mathbb{R} | f \text{ is linear}\}$. Each functional l_i is associated with a subentity of the reference element R .

The basis functions $\{\phi_0, \dots, \phi_{n-1}\}$ of the finite element space are defined by

$$l_i(\phi_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j. \end{cases} \quad (3.22)$$

To ensure continuity of the global finite element space, the functionals sharing subentities between neighbouring cells are assigned the same global degree-of-freedom.

The construction of a high-order degree-of-freedom map is illustrated by following an example from [46]. Consider a reference quadrilateral with 16 degrees-of-freedom, such that $\hat{l}_0, \dots, \hat{l}_3$ are associated with the vertices, $\hat{l}_4, \dots, \hat{l}_{11}$ are associated with the edges, and $\hat{l}_{12}, \dots, \hat{l}_{15}$ are associated with the cell.

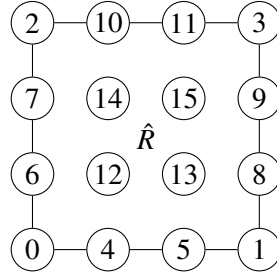
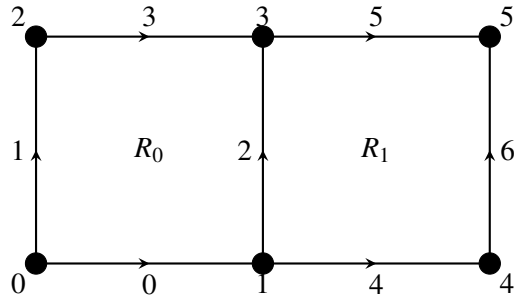


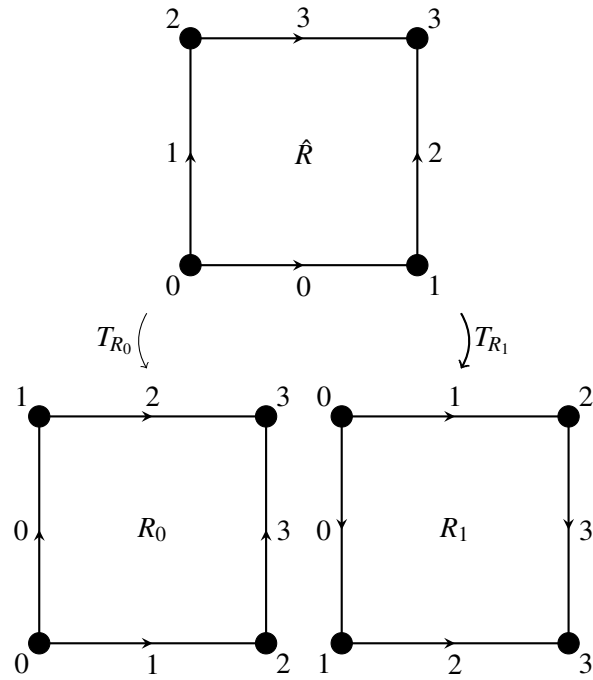
Fig. 3.3 Degree-of-freedom layout for degree 3 Lagrange element on the reference element.

Given an element degree-of-freedom layout, it is then possible to construct a finite element space V_h . Firstly, the global degrees-of-freedom based on the mesh topology are numbered, beginning with the degrees-of-freedom associated with the vertices, then the edges, then the facets, and finally the cells. Next, elements with shared entities are set to share the degrees-of-freedom between them. This defines a degree-of-freedom map for the space V_h , denoted $D_{V_h}(R) : \hat{L}_R \rightarrow R$, which maps the local degrees-of-freedom associated with cell R to the global degrees-of-freedom of the finite element space V_h .

As an example, consider the construction of a degree 3 Lagrange space over the mesh in figure 3.4a, consisting of cell R_0 and R_1 . Applying the process above, the finite element space as shown in figure 3.5 is obtained. Firstly, note that the global degrees-of-freedom at the vertices matches exactly across neighbouring cells. In particular, the reference vertices 0, 1, 2, and 3 are mapped to mesh vertices 0, 2, 1, and 3 for cell R_0 and reference vertices 0, 1, 2, and 3 are mapped to mesh vertices 3, 1, 5, and 4 for cell R_1 . However, at the edges, mismatch is observed for the global degrees-of-freedom l_{10} and l_{11} . This is due to the orientation of the edges. The local orientations of the edges are represented by the arrows in figure 3.4b. In this instance, the edges are arbitrarily oriented based on low-to-high vertex numbering, i.e., edge 0 is the edge from vertex 0 to vertex 1, edge 1 is the edge from vertex 0 to vertex 2, edge 2 is the edge from vertex 1 to vertex 3, and edge 3 is the edge from vertex 2 to vertex 3. Due to this particular numbering process, the orientation of the shared edges may disagree, resulting in inconsistencies of the global degrees-of-freedom on the edge. In two dimensions, disagreements



(a) The mesh with numbered vertex and edges. The arrows on the edges represents the edges orientation.



(b) The mapping of the reference cell to the mesh cells.

Fig. 3.4 A mesh and its finite element space construction from a reference cell.

between the orientation of neighbouring cell edges may occur. In three dimensions, in addition to edges, the faces may disagree between neighbouring cells. In fact, the disagreement between the orientation of the cell faces can include rotations as well as reflections [141].

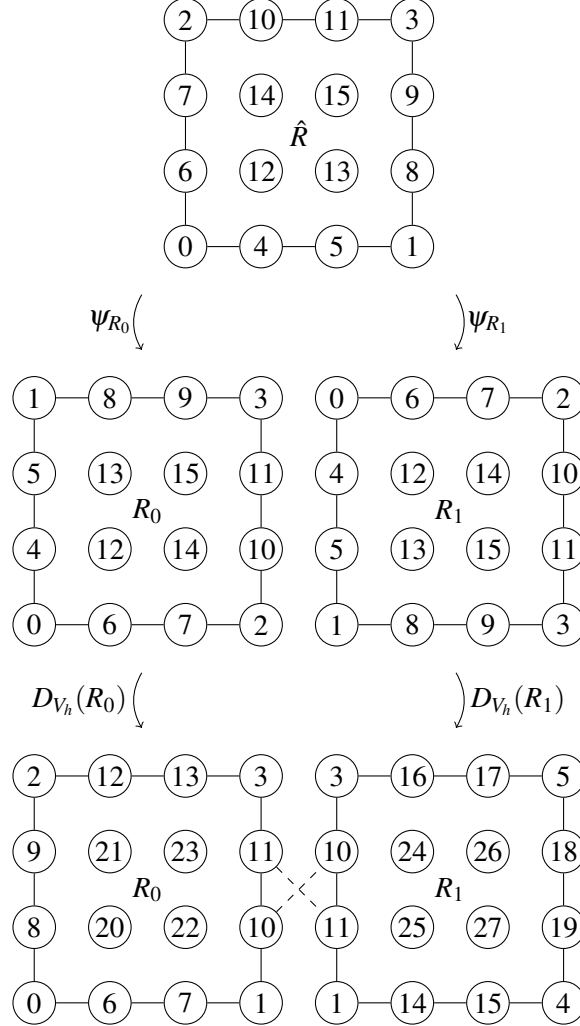


Fig. 3.5 Constructing the degree 3 Lagrange finite element space. Observed the mismatch of the global degrees-of-freedom l_{10} and l_{11} .

To correct for the disagreement of the sub-entity orientations, Scroggs et al. [141] developed an approach that applies a transformation to the degrees-of-freedom. This approach involves an intermediate step, whereby a new set of degree-of-freedom $\tilde{L}_R = \{\tilde{l}_{R,0}, \dots, \tilde{l}_{R,m-1}\}$ are constructed using linear combinations of $\hat{L}_R = \{\hat{l}_{R,0}, \dots, \hat{l}_{R,m-1}\}$ such that

$$\begin{pmatrix} \tilde{l}_{R,0} \\ \vdots \\ \tilde{l}_{R,m-1} \end{pmatrix} = M_R^{V_h} \begin{pmatrix} \hat{l}_{R,0} \\ \vdots \\ \hat{l}_{R,m-1} \end{pmatrix} \quad (3.23)$$

Applying the transformations $M_{R_0}^{V_h}$ and $M_{R_1}^{V_h}$ to $\hat{L}_0$ and $\hat{L}_1$, respectively, gives the transformed degrees-of-freedom $\tilde{L}_0$ and $\tilde{L}_1$. As a result, the correct orientation of the degrees-of-freedom on local edges 0 and 3 is obtained as illustrated in figure 3.6. It can be seen that the correct numbering of the global degrees-of-freedom is achieved between neighbouring cells.

It is remarked here that for scalar-valued spaces such as the Lagrange space, even though the transformation can be expressed as a matrix multiplication, in practice it is performed without forming $M_R^{V_h}$. In addition, since the transformation matrix for a Lagrange space are permutation matrices, it can be applied during the degree-of-freedom map construction, thus avoiding the application of the transformation matrix during assembly.

3.5 Solver algorithm

In this section, the steps taken by the solver to simulate a FUS are presented. These steps are illustrated in algorithm 1. The solver requires several inputs, such as a basis order P , a mesh, medium parameters and temporal parameters. Once these parameters are prescribed, the solver loops through the steps given in line 3 to 14 until the final time is reached. A key step of the algorithm is the Runge–Kutta step shown in line 5 to 12. Since the 4th order explicit Runge–Kutta method is used, the solver performs four loops, one for each stage of the Runge–Kutta step where the time-derivative are computed based on equation (3.15). Accordingly, for each iteration of the solver, four iterations are performed for the Runge–Kutta step.

Algorithm 1 Solver algorithm.

Input: P , mesh, ρ_0 , c_0 , δ , β

Output: $\mathbf{p}_{\text{final}}$, $\mathbf{v}_{\text{final}}$

```

1:  $\mathbf{p} \leftarrow \mathbf{p}_{\text{init}}$ ,  $\mathbf{v} \leftarrow \mathbf{v}_{\text{init}}$  ▷ Initialise vectors
2:  $t \leftarrow 0$ ,  $\mathbf{p}_1 \leftarrow 0$ ,  $\mathbf{v}_1 \leftarrow 0$ ,  $\mathbf{k}_1^P \leftarrow 0$ ,  $\mathbf{k}_1^V \leftarrow 0$ 
3: while  $t < t_{\text{end}}$  do
4:    $\Delta t \leftarrow \min(\Delta t, t_{\text{end}} - t)$ 
5:    $\mathbf{p}_0 \leftarrow \mathbf{p}$ ,  $\mathbf{v}_0 \leftarrow \mathbf{v}$ 
6:   for  $i = 1, \dots, 4$  do ▷ Runge–Kutta steps
7:      $\mathbf{p}_i \leftarrow \mathbf{p}_0$ ,  $\mathbf{v}_i \leftarrow \mathbf{v}_0$ 
8:      $\mathbf{p}_i \leftarrow \mathbf{p}_i + a_i \mathbf{k}_i^P \Delta t$ ,  $\mathbf{v}_i \leftarrow \mathbf{v}_i + a_i \mathbf{k}_i^V \Delta t$  ▷ Compute intermediate vectors
9:      $t_i \leftarrow t + c_i \Delta t$  ▷ Compute intermediate time
10:    Compute  $\mathbf{k}_i^P$  based on equation (3.15a)
11:    Compute  $\mathbf{k}_i^V$  based on equation (3.15b)
12:     $\mathbf{p} \leftarrow \mathbf{p} + b_i \mathbf{k}_i^P \Delta t$ ,  $\mathbf{v} \leftarrow \mathbf{v} + b_i \mathbf{k}_i^V \Delta t$  ▷ Update solutions
13:   end for
14:    $t \leftarrow t + \Delta t$ 
15: end while
16:  $\mathbf{p}_{\text{final}} \leftarrow \mathbf{p}$ ,  $\mathbf{v}_{\text{final}} \leftarrow \mathbf{v}$ 

```

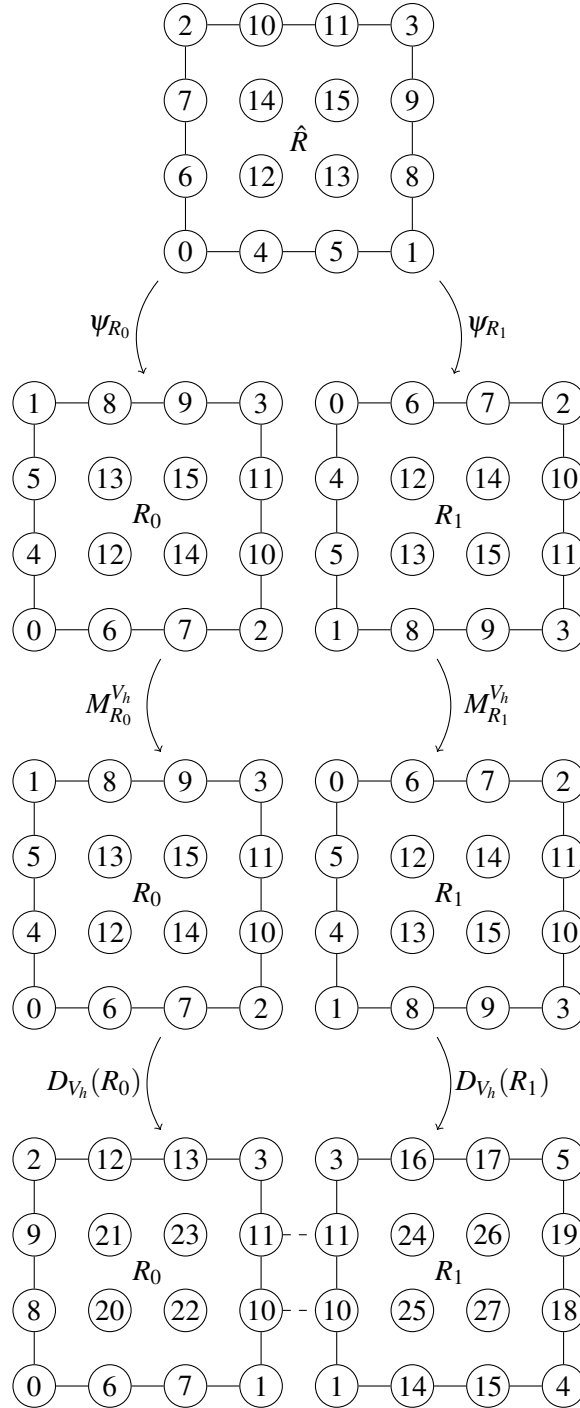


Fig. 3.6 The degrees-of-freedom layout obtained after degrees-of-freedom transformation.

3.6 Summary

In this chapter, the numerical methods for the space and time discretisation of the continuous IBVP of equation (2.37) were presented. The 4th order explicit Runge–Kutta method is utilised for the temporal discretisation and the mass lumped finite element method is used for the spatial discretisation. The combination of these two numerical methods leads to a reduction in computational cost, since the solving of a linear system is avoided. The degree-of-freedom map that enables the definition of arbitrary degree of finite element basis on quadrilateral and hexahedral meshes is also discussed.

Chapter 4

Implementation of the numerical methods

The finite element and Runge–Kutta methods presented in chapter 3 provide the basis of the numerical method used in this thesis. In this chapter, the computer implementation of the numerical methods discussed in chapter 3 is presented. Additionally, the most costly operations in the solver’s implementation are identified.

First, in section 4.1, a summary of the FEniCSx software is presented. The software provides the interface for the implementation of the finite element method. Next, in section 4.2, the parallelisation that is built in FEniCSx is presented, and in particular, the MPI parallelism that enables FEniCSx to be used for large-scale simulation. In section 4.3, an overview of the computer implementation of the numerical methods discussed in chapter 3 is presented. The C++ API of the FEniCSx software is utilised and the solver design presented. Accordingly, in section 4.4, the Unified Form Language (UFL) is presented, along with a description of its contents. In particular, UFL demonstrates how to define the GLL basis function as well as the GLL quadrature in FEniCSx. Subsequently, in section 4.5, several key implementations in the solver are presented, which includes assembly of the global vectors and the Runge–Kutta functions evaluation. In section 4.6, the computational bottlenecks of the implementation are identified.

Sections 4.1 and 4.2 contain implementations that are readily available in FEniCSx and are presented for completeness. Sections 4.3 to 4.6 contain implementations by the author.

4.1 FEniCSx software

FEniCSx is an open-source software library for solving partial differential equations. The development of FEniCSx started in 2018 to address issues in legacy FEniCS libraries and to leverage new supporting tools [2]. FEniCSx provides several improvement as compared to legacy FEniCS such as the support for different floating-point precision, complex numbers, the support for quadrilateral and hexahedral elements and most importantly, a focus on parallelism from the outset. As a result, this provides

FEniCSx users with the capability to leverage high-performance computing platforms. The FEniCSx software provides both Python and C++ interfaces.

FEniCSx consists of a collection of libraries, namely UFL [14], Basix [140], FFCx, and DOLFINx [20]. The UFL (Unified Form Language) library is a domain-specific language for weak formulations of partial differential equations. In other words, UFL provides the user with an interface to define weak forms in a notation that is close to the mathematical notation. Additionally, it gives the user the option to select the finite element spaces. The Basix library is a finite element definition and tabulation runtime library. In the context of the finite element method, Basix provides several capabilities such as the evaluation of finite element basis functions and their derivatives at a set of points, the mapping between physical and reference cell, and the permutation of degrees-of-freedom for high-order finite elements. The FFCx (FEniCS form compiler) library is a compiler for finite element variational form. More precisely, FFCx compiles the variational form that is defined by UFL and generates the low-level C code that contains performant finite element kernels. The DOLFINx library is the computational backend of the FEniCSx software that provides the high-performance parallel processing capability. Specifically, DOLFINx handles the computation of the global degrees-of-freedom of the finite element problem. This includes the reading of the mesh, the generation of finite element function spaces, the assembly of the linear and bilinear forms, and the solving of the system of equations. DOLFINx also implements the distributed memory parallelism capability of FEniCSx. Figure 4.1 shows the component of FEniCSx open-source software.

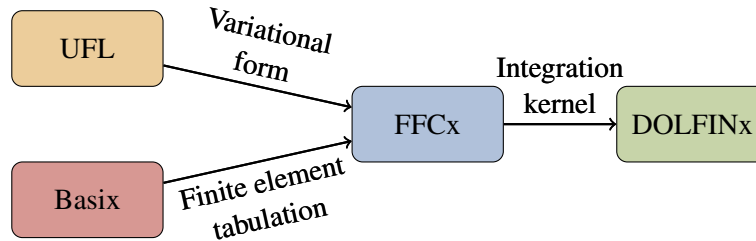


Fig. 4.1 The components of the FEniCSx open-source software.

4.2 Distributed memory parallelism

Before proceeding with the computer implementation of the numerical methods, a brief presentation of the parallelism in DOLFINx is discussed. This feature is important for the solver, as it is designed to simulate realistic application settings, which typically involve large 3D simulations.

In parallel, the degrees-of-freedom (DOF) of the finite element solver needs to be distributed across different processes. Each global DOF must be owned by a single process. While some DOF may be shared with non-owning processes. Within a particular process, shared DOF that is owned by another process is called a *ghost* DOF, as illustrated in figure 4.2.

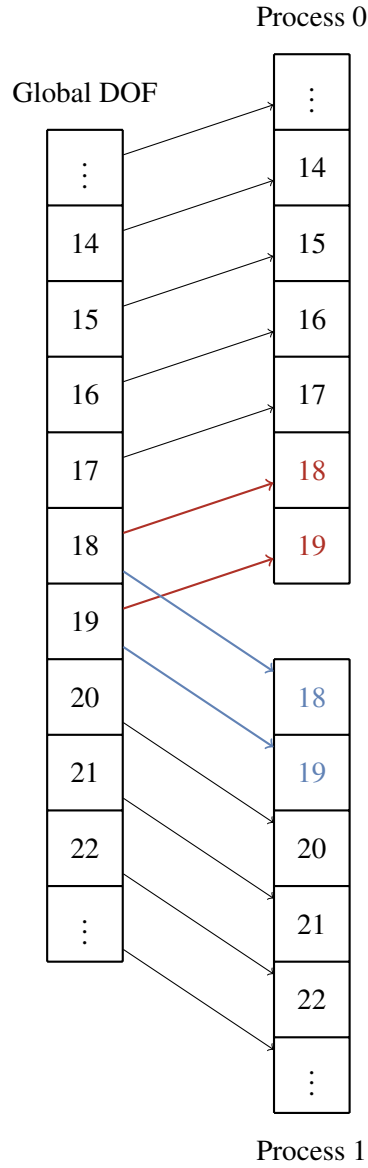


Fig. 4.2 Illustration of the owned and ghost degrees-of-freedom. When running in parallel, DOF 0 to 19 is send to process 0 and DOF 18 to DOF N is send to process 1, where N is the global number of degrees-of-freedom. Both processes contain DOF 18 and 19. However, DOF 18 and 19 in process 0 are *ghost* DOF (in red), while process 1 owns DOF 18 and 19 (in blue).

In DOLFINx, the distribution of the DOF across processes is determined by and stored in an `IndexMap` object. The `IndexMap` object store data and provide several functionalities. Most importantly, the `IndexMap` object distributes the DOF across the processes. Each process then owns a contiguous list of DOF as well as a non-contiguous list of ghost DOF. Additionally, the object stores the owning rank of each ghost DOF, as well as the source and destination rank. The source ranks are the processes that own the ghost DOF while the destination ranks are the processes that contain the ghost DOF. Ultimately, the `IndexMap` object enables precise and error-free communications between different processes. The principal communications in a parallel finite element solver are scatter forward and scatter reverse.

4.2.1 Scatter forward

Scatter forward function sends data associated with owned indices to ghosting ranks. In other words, scatter forward takes data from the processes that owns the data to their associated ghost data in other processes. This mechanism is illustrated in figure 4.3.

In DOLFINx, the scatter forward function is implemented as the `scatter_fwd` function. The `scatter_fwd` function is typically used to update ghost DOF in a finite element function. The scatter forward function uses an implementation that is designed for sparse communication patterns [32, 78], which is particularly suited for parallel finite element implementation. Accordingly, this ensures optimal performance for the MPI communications.

4.2.2 Scatter reverse

In contrast to scatter forward, scatter reverse sends data from the ghost indices to the owned ranks. In particular, the scatter reverse takes the ghost data from ghost ranks and send it to the owning ranks. This mechanism is illustrated in figure 4.4.

In DOLFINx, the scatter reverse function is implemented as the `scatter_rev` function. The `scatter_rev` function is usually use after the assembly process, i.e., to add the contributions of the ghost DOF to the owning DOF. Like the scatter forward function, the scatter reverse function also uses an implementation that is designed for sparse communication patterns [32, 78], namely the `MPI_Neighbor_alltoall` function.

4.3 Implementation overview

In this work, the C++ programming language is used, as it provides more control and access over low-level functions in DOLFINx, which simplify modification and gives better computational performance. Moreover, a *naive*¹ implementation of the Runge–Kutta method in C++ gives better performance than in Python.

¹a simple and direct implementation without any language-specific optimisations.

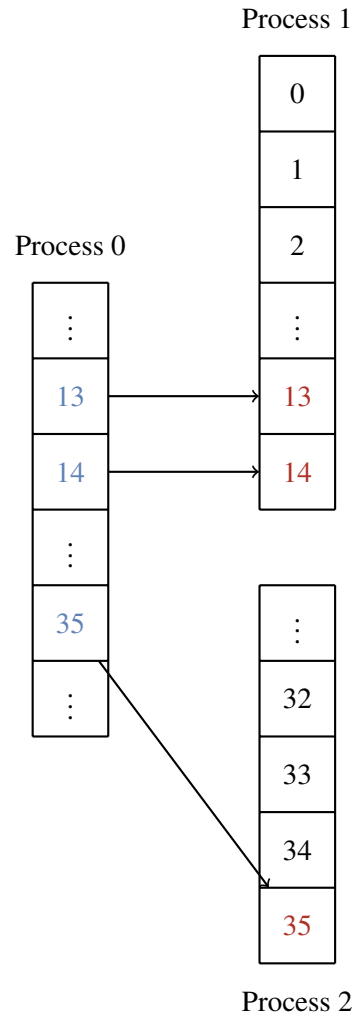


Fig. 4.3 The scatter forward mechanism. In this instance, process 0 owns DOF 13, 14, and 35 (shown in blue). Process 1 contains DOF 13 and 14 as ghost DOF and process 2 contains DOF 35 as ghost DOF (shown in red). The scatter forward operation send data from owned DOF to ghost DOF. In this regard, the data from DOF 13, 14, and 35 in process 0 are send to DOF 13 and 14 in process 1 and DOF 35 in process 2. The direction of the data transfer is represented by the arrows.

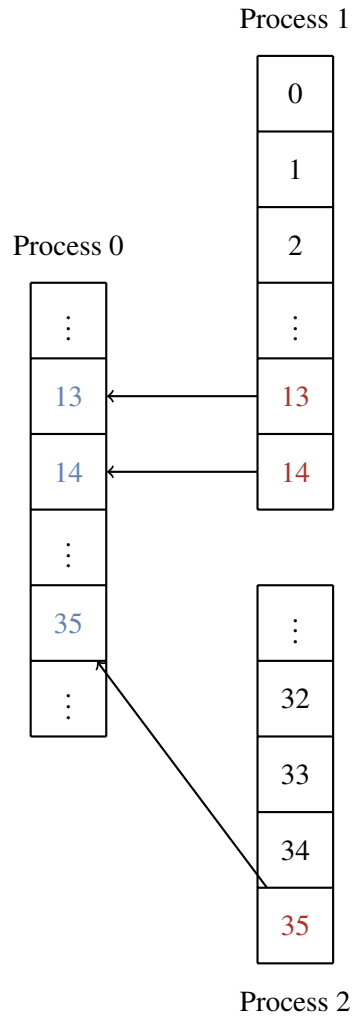


Fig. 4.4 The scatter reverse mechanism. In this example, process 0 owns DOF 13, 14, and 35 (shown in blue). Process 1 contains DOF 13 and 14 as ghost DOF and process 2 contains DOF 35 as ghost DOF (shown in red). The scatter reverse operation send data from ghost DOF to owned DOF. In this regard, the data from the ghost DOF 13 and 14 of process 1 and ghost DOF 35 from process 2 is send to process 0. The direction of the data transfer is represented by the arrows.

To start, the implementation requires a Python file that represents the UFL input, which defines the relevant linear and bilinear forms. This file must contain the coordinate element of the mesh and the finite element type. Additionally, it must contain the definition of the function space, the relevant functions, as well as the definition of the corresponding linear and bilinear forms. The selection of the finite element type is done using Basix while the linear and bilinear forms are defined using UFL. The input file is then compiled using FFCx, which generates two files namely, a .c file and a .h file. The .c file contains the declaration of the functions for the finite element kernels while the .h file—a header file—contains its definition. The functions in the generated files are then used in the solver file called `Westervelt.hpp`. The `Westervelt.hpp` file contains the bulk of the implementation such as the creation of the *global* function space and functions, the initialisation of the initial conditions for p_h and v_h , the definition of k_i^p and k_i^v based on equation (3.15), the assembly of the global vectors, the scaling of vector, as well as the Runge–Kutta implementation. Figure 4.5 visualises the compilation process by FFCx and the usage of the generated files by `Westervelt.hpp`.

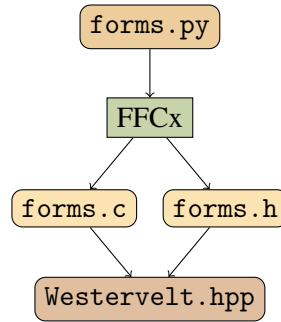


Fig. 4.5 Visualisation of a FFCx compilation for `forms.py` and the generated code that is used by `Westervelt.hpp`.

4.4 The UFL problem description

In this section, the details of the UFL file described in section 4.3 are discussed. In particular, the definitions of the linear and bilinear forms of the discrete variational form described in equation (3.9b) are presented. Most importantly, the process of defining a continuous Lagrange finite element with GLL interpolation points is presented. Additionally, the GLL quadrature scheme is introduced to ensure the generation of a diagonal mass matrix. To represent region with different physical properties, the definition of a zeroth-order discontinuous Galerkin (DG0) finite element is presented.

The UFL file begins with the definition of the coordinate element of the mesh and is defined as follows

```

from ufl import VectorElement, Mesh, hexahedron

coord_element = VectorElement("Lagrange", hexahedron, 1)
mesh = Mesh(coord_element)

```

The code above defines a first order geometry, i.e, cells with straight sides. While it is possible to define higher order geometry in FEniCSx, this work is restricted to first order meshes. Higher order mesh can simply be defined just by changing the input to the `VectorElement` function. Next, the definition of a continuous Lagrange finite element with GLL points is defined as follows

```
import basix.ufl

element_family = basix.ElementFamily.P
cell_type = basix.CellType.hexahedron
degree = 4
variant = basix.LagrangeVariant.gll_warped

element = basix.ufl.create_element(element_family, cell_type, degree, variant
)
```

In this example, a P4 Lagrange finite element on a hexahedral cell with GLL interpolation points is defined. It is remarked here that the solver is not only restricted to P4 elements (see section 6.3). Once the mesh and element have been defined, a function space and functions on the space are defined such that

```
from ufl import FunctionSpace, Coefficient

# Define function space
V = FunctionSpace(mesh, element)

# Define functions
vdot = Coefficient(V)
p = Coefficient(V)
v = Coefficient(V)
g = Coefficient(V)
dg = Coefficient(V)
w = TestFunction(V)
```

It is highlighted here that the variable `vdot` is defined as a `Coefficient` instead of as a `TrialFunction`. This is done to avoid the assembly of a matrix, given that the mass matrix is known *a priori* to be a diagonal matrix. Note that the variables defined above correspond exactly to each of the functions that appear in equation (3.9b). To handle the heterogeneity of the domain, a DG0 element is defined such that

```
element_DG = basix.ufl.create_element(
    element_family, cell_type, 0, variant)

# Define DG function space
V_DG = FunctionSpace(mesh, element_DG)

# Define DG functions
c0 = Coefficient(V_DG)
rho0 = Coefficient(V_DG)
```

```
delta = Coefficient(V_DG)
beta = Coefficient(V_DG)
```

Note that the physical properties of the domain (speed of sound, density, diffusivity of sound, coefficient of nonlinearity) is defined using DG0 function space. In other words, the heterogeneity of the domain has cell-wise variation, which is a common approach to define domain heterogeneity [160, 69]. Finally, to implement the mass lumped scheme, the appropriate quadrature scheme and degree need to be set. This can be achieved as follows

```
# Map from quadrature points to basix quadrature degree
qdegree = {3: 4, 4: 5, 5: 6, 6: 8, 7: 10, 8: 12, 9: 14, 10: 16}
md = {"quadrature_rule": "GLL", "quadrature_degree": qdegree[Q]}
```

The Python dictionary above maps the number of quadrature points to the Basix quadrature degree that is required to ensure that Basix uses the correct number of quadrature points for a specific polynomial basis degree. Thus, ensuring that a diagonal mass matrix is obtained. Currently in FEniCSx, this only works with tensor product element such as quadrilateral and hexahedral element. Once these are established, the forms can be defined, which is restated here (the subscript h has been removed and $\dot{v} = \partial v / \partial t$ defined for clarity)

$$\begin{aligned} \int_{\Omega} \left(\frac{1}{\rho_0 c_0^2} - \frac{2\beta}{\rho_0^2 c_0^4} p \right) \dot{v} w \, d\Omega + \int_{\Gamma} \frac{\delta}{\rho_0 c_0^3} \dot{v} w \, d\Gamma = & - \int_{\Omega} \frac{1}{\rho_0} \nabla p \cdot \nabla w \, d\Omega \\ & - \int_{\Gamma} \frac{1}{\rho_0 c_0} v w \, d\Gamma - \int_{\Omega} \frac{\delta}{\rho_0 c_0^2} \nabla v \cdot \nabla w \, d\Omega + \int_{\Omega} \frac{2\beta}{\rho_0^2 c_0^4} v^2 w \, d\Omega \\ & + \int_{\Gamma_s} \left(\frac{1}{\rho_0} g + \frac{\delta}{\rho_0 c_0^2} \frac{\partial g}{\partial t} \right) w \, d\Gamma_s, \quad (4.1) \end{aligned}$$

such that

```
# LHS form
a = inner(1/(rho0*c0**2) - (2*beta0*p/(rho0**2*c0**4)) * vdot, w)
    * dx(metadata=md)
    + inner(delta0/(rho0*c0**3) * vdot, w) * ds(metadata)

# RHS form
L = - inner(1/rho0 * grad(p), grad(w)) * dx(metadata=md)
    - inner(1/(rho0*c0) * v, w) * ds(metadata=md)
    - inner(delta0/(rho0*c0**2) * grad(v), grad(w)) * dx(metadata=md)
    + inner(2*beta0/(rho0**2*c0**4) * v*v, w) * dx(metadata=md)
    + inner(1/rho0 * g + delta0/(rho0*c0**2) * dg, w) * ds(1, metadata=md)

forms = [a, L]
```

Listing 4.1 Definition of the left-hand side and right-hand side forms of equation (4.1). Note that the forms definition maps exactly to the equation.

The variable `ds` denotes the entire domain boundary and `ds(1)` denotes the source boundary. Observe that the above forms are similar to the terms contain in equation (4.1), which show that UFL provides

an interface to define weak forms in a notation close to mathematical notation. Finally, the UFL file is compiled using FFCx using the following command

```
ffcx forms.py --scalar_type=double
```

which generates the `forms.c` and `forms.h` files. These files contain the definition and declaration of the finite element kernels. In particular, FFCx takes the data computed by Basix and generates the functions to compute and assemble the local vector defined by the forms. Note that the `--scalar_type=double` option is passed to FFCx. Alternatively, it is also possible to pass different floating-point precision such as `single`, `complex` or `complex double`. A choice that is discussed in section 5.4.

4.5 The FEniCSx-FUS solver

The implementation of a computationally efficient solver depends on several aspects such as the efficiency of the algorithm as well as a rigorous implementation. Additionally, for large-scale simulation, the scalability of the solver is important. Most of these aspects are already supported by the FEniCSx software. In this section, the key implementations of the solver using the FEniCSx software are presented.

4.5.1 Assembly of the global diagonal mass *matrix*

A key aspect of the efficiency of the solver is due to the mass lumped scheme that results in a diagonal mass matrix. To reduce the assembly cost, the assembly of a global matrix is avoided. Instead, a vector is assembled since it is known *a priori* that the mass matrix only contains diagonal entries. To assemble a vector consisting of the diagonal entries of the mass matrix, the function space and function needs to be defined as follows

```
// Define function space
V = std::make_shared<fem::FunctionSpace>(
    fem::create_function(functionspace_form_forms_a, "u", mesh));

// Define function
vdot = std::make_shared<fem::Function<T>>>(V);
```

Once the `vdot` function has been defined, the function is filled with ones such that

```
std::span<T> vdot_ = vdot->x()->mutable_array();
std::fill(vdot_.begin(), vdot_.end(), 1.0);
```

This step is a **key** step since a vector is being assembled instead of a matrix. Typically, the `vdot` function is a `TrialFunction`, which results in the assembly of a matrix. But, since the GLL interpolation points are used and the appropriate GLL quadrature degree is utilised, it is mathematically equivalent, i.e. the assembled vector contains the entries of the diagonal mass matrix. Finally, the global vector is assembled as follows

```

// Define form
a = std::make_shared<fem::Form<T>>(<
    fem::create_form<T>(*form_forms_a, {V},
        {"vdot", vdot}, {"p", p}, {"c0", c0}, {"rho0", rho0}, {"delta0",
        delta0}
        {"beta0", beta0}},
    {}),
    {{dolfinx::fem::IntegralType::exterior_facet, &(*ft)}}));

// Assemble vector
index_map = V->dofmap()->index_map; // Index map
bs = V->dofmap()->index_map_bs(); // Index map block size

m = std::make_shared<la::Vector<T>>(index_map, bs);
m_ = m->mutable_array();
std::fill(m_.begin(), m_.end(), 0.0);
fem::assemble_vector(m_, *a);

```

The code above assemble the left-hand side of equation (4.1). The code listings above illustrate the simple interface provided by DOLFINx to assemble the global vector that contains the entries of the diagonal mass matrix.

4.5.2 Evaluation of the Runge–Kutta time-derivative functions

The two main functions of the solver are the computations of the intermediate slopes of the Runge–Kutta steps, namely the $\mathbf{k}^p$ and $\mathbf{k}^v$ computations based on equation (3.15). The evaluation of $\mathbf{k}^p$ in equation (3.15a) is implemented as follows

```

/// Evaluate  $\mathbf{k}^p = \mathbf{f}(\mathbf{p}, \mathbf{v})$ 
void fp(T& t, std::shared_ptr<la::Vector<T>> p,
    std::shared_ptr<la::Vector<T>> v,
    std::shared_ptr<la::Vector<T>> result) {
    copy<T>(*v, *result);
}

```

Listing 4.2 $\mathbf{k}^p$ vector evaluation.

Evidently, the function above consist of only the copy function. In other words, the evaluation of $\mathbf{k}^p$ returns $\mathbf{v}$. The evaluation of $\mathbf{k}^v$ in equation (3.15b) is implemented as follows

```

/// Evaluate  $\mathbf{k}^v = \mathbf{f}(\mathbf{v}, \mathbf{u})$ 
void fv(T& t, std::shared_ptr<la::Vector<T>> p,
    std::shared_ptr<la::Vector<T>> v,
    std::shared_ptr<la::Vector<T>> result) {

    // Update source
    std::fill(g_.begin(), g_.end(), 2.0 * p0 * w0 / s0 * cos(w0 * t));
    std::fill(dg_.begin(), dg_.end(),
        2.0 * p0 * w0 / s0 * cos(w0 * t)

```

```

        - 2.0 * p0 * w0 * w0 / s0 * sin(w0 * t));

// Update fields
pi->scatter_fwd();
copy<T>(*pi, *p->x());

vi->scatter_fwd();
copy<T>(*vi, *v->x());

// Assemble LHS
std::fill(m_.begin(), m_.end(), 0.0);
fem::assemble_vector(m_, *a);
m->scatter_rev(std::plus<T>());

// Assemble RHS
std::fill(b_.begin(), b_.end(), 0.0);
fem::assemble_vector(b_, *L);
b->scatter_rev(std::plus<T>());

// Scaling
{
    out = result->mutable_array();
    _b = b->array();
    _m = m->array();

    // out[i] = b[i]/m[i]
    std::transform(_b.begin(), _b.end(), _m.begin(), out.begin(),
        [](const T& bi, const T& mi) { return bi / mi; });
}
}

```

Listing 4.3 $\mathbf{k}^v$ vector evaluation.

The evaluation of $\mathbf{k}^v$ consists of several steps. These steps include updating of the source functions ($g(t)$ and $\partial g(t)/\partial t$), updating the field functions (p and v), reassembly of the forms, and a vector scaling operation (*solve* step). The evaluation of $\mathbf{k}^v$ also involves MPI communications using the scatter forward and scatter reverse functions discussed in section 4.2.

4.6 Identifying solver bottleneck

A detail investigation of the FEniCSx-FUS implementation is performed to identify the bottlenecks within the solver. On this note, the areas that are computationally expensive are demonstrated through simulation data. A top-to-bottom approach is used starting from the time-stepping implementation of listing 4.4. Next, the time-derivative f^v function of listing 4.3 is studied, and finally, the assembly of each terms in listing 4.1 is analysed.

```

while (t < tf)
{
    dt = std::min(dt, tf - t);

    // Store temporary vectors 1 (COPY 1)
    kernels::copy<T>(*p_, *p0);
    kernels::copy<T>(*v_, *v0);

    // Runge--Kutta 4th order step
    for (int i = 0; i < 4; i++) {

        // Store temporary vectors 2 (COPY 2)
        kernels::copy<T>(*p0, *pi);
        kernels::copy<T>(*v0, *vi);

        // Compute intermediate vectors (AXPY A)
        kernels::axpy<T>(*pi, dt * a_runge[i], *kp, *pi);
        kernels::axpy<T>(*vi, dt * a_runge[i], *kv, *vi);

        // Compute intermediate time
        tn = t + c_runge[i] * dt;

        // Time-derivative evaluations
        fu(tn, pi, vi, kp);
        fv(tn, pi, vi, kv);

        // Update solutions (AXPY B)
        kernels::axpy<T>(*p_, dt * b_runge[i], *kp, *p_);
        kernels::axpy<T>(*v_, dt * b_runge[i], *kv, *v_);
    }
}

```

Listing 4.4 Time-stepping code. The implementation of the solver algorithm of algorithm 1. Observed that the implementation maps exactly to the algorithm. For conciseness, each operations are labelled using shorter names, e.g., the store temporary vectors 1 is labelled as COPY 1.

Listing 4.4 shows the implementation of algorithm 1. There are four functions within the algorithm, namely the copy, axpy, fp, and fv functions. The timings of these functions are shown in figure 4.6. The bulk of the time spent within the algorithm is in the fv function that is close to 97% of the total time. This is unsurprising since the fv function contains several operations including the copy and axpy functions. On the other hand, the copy and axpy functions are comparatively cheap where they constitutes between 0.2% to 1.0% of the time. Recall in section 4.5, the fp function contains only a single copy function, thus, is also comparatively cheap.

Figure 4.7 shows the percentage of time spent by each function within the fv function. Indeed, it is observed that the highest percentage of time spent within the fv function is in the assembly of the right-hand side vector, which is approximately 65%. The second highest percentage is the assembly

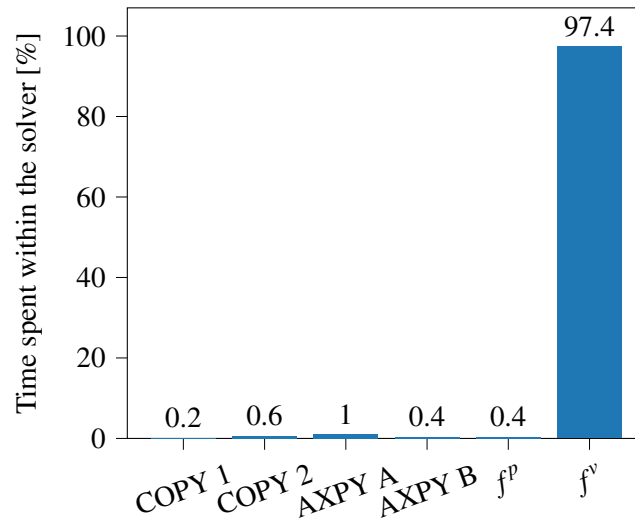


Fig. 4.6 The percentage of time spent within each functions in listing 4.4. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using $P4$ Lagrange element.

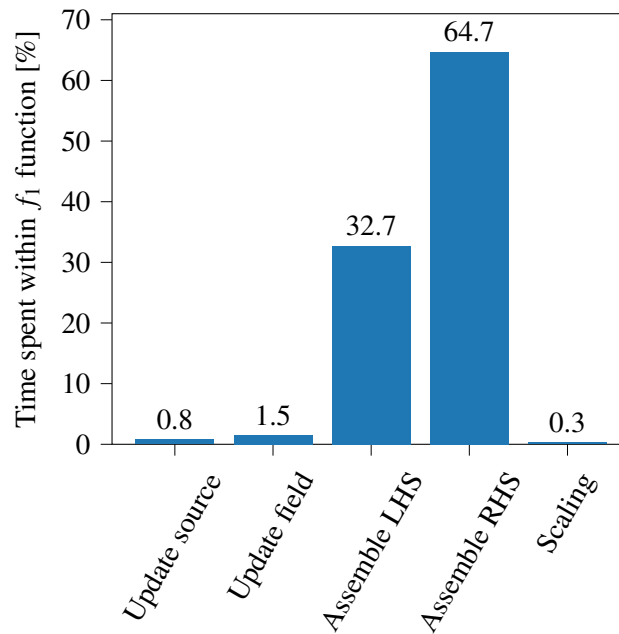


Fig. 4.7 The percentage of time spent in the f_v function of listing 4.3. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using $P4$ Lagrange element.

of the left-hand side vector, which is around 33%. The cheapest operation is the solve step where the element-wise division between two vectors is performed. In fact, the solve step only constitutes 0.3% of the total time spent. Additionally, the cost of the update source step is small since it involves updating the source vectors, which is essentially equivalent to the copy function. Finally, the third highest percentage is the update field step that involves updating the field vectors. This step includes an MPI process and as such incurs additionally computational cost due to the communication between processes. Nonetheless, the most expensive step in the $\mathbf{fv}$ function is the assembly of the vectors.

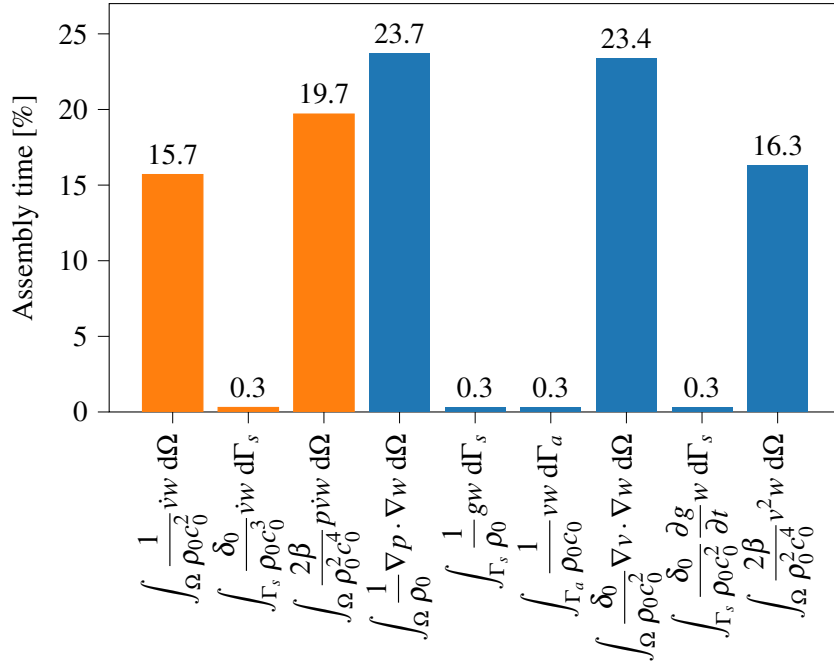


Fig. 4.8 The percentage of time spent for vector assembly of each terms in the forms definition of listing 4.1. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using $P4$ Lagrange element. The total time spent is 0.76 seconds. The orange bars represent the terms on the left-hand side and the blue bars represent the terms on the right-hand side.

Figure 4.8 shows the time spent to assemble each term that appears in the forms definition of listing 4.1. Evidently, the highest percentage of time is spent in the assembly of the terms that contains cell-wise computation, i.e., the integration over the cell elements $d\Omega$. In fact, approximately 99% of the total time is spent in the cell-wise computations. This is expected since the number of surface-wise computation is significantly less than the cell-wise computation— ds is significantly less than dx . Additionally, the surface-wise computation involves less number of degrees-of-freedom. With regard to the cell-wise computations, the terms involving the gradient operator are more computationally expensive.

4.7 Summary

In this chapter, the numerical implementation of the solver algorithm in algorithm 1 was presented. More specifically, the 4th order Runge–Kutta method and the mass lumped finite element method discussed in chapter 3 were implemented. The mass lumped finite element method was implemented using the FEniCSx open-source software. The FEniCSx open-source software supports arbitrary degree finite element and distributed memory parallelism. Two features that enable accurate, large-scale acoustic simulation in the FEniCSx-FUS solver. Additionally, the FEniCSx-FUS solver was analysed to identify its computational bottlenecks.

Chapter 5

Fast finite element integration

In chapter 4, the implementation of the FEniCSx-FUS solver was presented. In addition, the computational bottlenecks of the solver were identified and it was found that the bulk of the solver computation is spent in cell-wise finite element assembly. In this chapter, implementations that accelerate the cell-wise finite element assembly process are presented. Furthermore, the performance improvements that result from these implementations are analysed.

In section 5.1, a brief review of the finite element vector assembly process that is relevant to the solver is presented, i.e., the evaluations of the local mass operator and the local stiffness operator. It is highlighted that the evaluation of these operators involves the computation of integrals using numerical quadrature. Next, in section 5.2, an implementation to precompute the geometric data is introduced. Since the geometric data is typically unchanged for a focused ultrasound simulation, it is reasonable to compute and store the geometric data instead of recomputing it at each time step. Subsequently, in section 5.3, the sum factorisation algorithm and its implementation are presented. By exploiting the tensor-product structure of the mesh, the number of floating-point operations of the finite element integration are reduced. In section 5.4, to further enhance the solver performance, the implementation of reduced precision is presented. Finally, in section 5.5, the performance achieved by these implementations is demonstrated.

5.1 Cell-wise finite element assembly

In section 4.6, it was shown that the main bottleneck of the solver is the cell-wise vector assembly. In this section, a brief review of the cell-wise vector assembly process is presented. In particular, the assembly of the mass and the stiffness operator.

5.1.1 Evaluation of the local mass operator

Consider the local mass operator evaluation. In particular, the local mass operator evaluation that corresponds to the mass lumped finite element method. On a particular element, the local mass

operator evaluation is given as

$$m_j^k = \int_{\Omega_k} \left(\sum_{i=0}^{N_p} u_i \phi_i^k \right) \phi_j^k \, dx, \quad (5.1)$$

where N_p is the number of the local degrees-of-freedom, u_i are the degrees-of-freedom, and ϕ^k is the basis function. The symbol m_j^k represents the entries of the local mass *matrix*. The contributions of the local mass *matrices* constitutes the entries of the global mass *matrix* $\mathbf{M}$ defined in section 3.2. Recall that the local vector contribution is then added to the global vector based on the degrees-of-freedom map presented in section 3.4. As mentioned in section 3.3, the local vector computation is often performed on a reference element. Mapping the computation onto the reference element yields

$$m_j^k = \int_{\hat{\Omega}_k} \left(\sum_{i=0}^{N_p} u_i \phi_i^k(\hat{x}) \right) \phi_j^k(\hat{x}) |J_k| \, d\hat{x}, \quad (5.2)$$

where J_k is the Jacobian of the transformation between the physical and reference element and $|J_k|$ is its determinant. The hat symbol denotes the reference element variables and $\phi_i(\hat{x})$ denotes the reference element basis functions. Computing the integral using numerical quadrature results in

$$m_j^k = \sum_{q=0}^{N_q} \left(\sum_{i=0}^{N_p} u_i \phi_i^k(\hat{x}_q) \right) \phi_j^k(\hat{x}_q) w_q |J_k(\hat{x}_q)|, \quad (5.3)$$

where $\hat{x}_q$ and w_q are the quadrature points and weights, respectively. The symbol N_q denotes the number of quadrature points. Since GLL quadrature points are used as the basis function interpolation points, the equation reduces to

$$m_j^k = u_q w_q |J_k(\hat{x}_q)|. \quad (5.4)$$

Note that in equation (5.4), the number of computations is reduced by eliminating the j and q loops.

5.1.2 Evaluation of the local stiffness operator

Consider the local stiffness operator evaluation such that

$$b_j^k = \int_{\Omega_k} \sum_{i=0}^{N_p} u_i \nabla \phi_i^k \cdot \nabla \phi_j^k \, dx. \quad (5.5)$$

On the reference element, the equation above can be written as

$$b_j^k = \int_{\hat{\Omega}_k} \left(J_k^{-1} \sum_{i=0}^{N_p} u_i \nabla_{\hat{x}} \phi_i^k(\hat{x}) \right) \cdot \left(J_k^{-1} \nabla_{\hat{x}} \phi_j^k \right) |J_k| \, d\hat{x}, \quad (5.6)$$

where $\nabla_{\hat{x}}$ denotes the reference element gradient of reference element basis functions $\phi_i(\hat{x})$. The integral evaluation, performed using numerical quadrature, yields

$$b_j^k = \sum_{q=0}^{N_q} \left(J_k^{-1}(\hat{x}_q) \sum_{i=0}^{N_p} u_i \nabla_{\hat{x}} \phi_i^k(\hat{x}) \right) \cdot \left(J_k^{-1}(\hat{x}_q) \nabla_{\hat{x}} \phi_j^k(\hat{x}) \right) w_q |J_k(\hat{x}_q)|. \quad (5.7)$$

Equations (5.4) and (5.7) show that the assembly of a local vector consists primarily of the evaluation of finite element integrals. Thus, a fast finite element integral evaluation can accelerate the vector assembly process. Two approaches are employed for fast finite element integration, namely by precomputing and storing the geometric data (see section 5.2), and by using fast numerical quadrature algorithm (see section 5.3).

5.2 Precomputing geometric data

In section 5.1, it was shown that the evaluations of the local mass operator of equation (5.4) and the local stiffness operator of equation (5.7) require the computation of geometric data, e.g., the Jacobian determinant $|J_k|$. For a fixed mesh, the geometric data such as the Jacobian and its determinant are constant for the entire simulation. As such, by precomputing the geometric data, the number of floating-point operations of the solver at each Runge–Kutta time step can be reduced. In this section, the concept of precomputing the geometric data is presented.

In practice, the mesh of a FUS simulation is typically fixed, i.e., the mesh structure does not change throughout the entire simulation. As a result, the geometric data such as the Jacobian and its determinant can be precomputed and stored. To this end, rewriting equation (5.4) such that

$$m_q^k = u_q C(\hat{x}_q) \quad (5.8)$$

where $C(\hat{x}_q) = w_q |J_k(\hat{x}_q)|$. Hence, in each cell and at each quadrature point, the variable $C(\hat{x}_q)$ is stored. Consequently, the memory needed for storing $C = \{C(\hat{x}_0), C(\hat{x}_1), \dots\}$ amounts to $n_{\text{cell}} \times N_q$, where n_{cell} represents the number of cells in the mesh. In practice, this quantity is relatively small compared to the total number of degrees-of-freedom.

Consider equation (5.7), rewriting the equation such that

$$b_j^k = \sum_{q=0}^{N_q} \left(\sum_{i=0}^{N_p} u_i \nabla_{\hat{x}} \phi_i^k(\hat{x}_q) \right) G(\hat{x}_q) \nabla_{\hat{x}} \phi_j^k(\hat{x}_q) \quad (5.9)$$

where $G(\hat{x}_q) = J_k^{-1}(\hat{x}_q) J_k^{-T}(\hat{x}_q) C(\hat{x}_q)$. Here, $G(\hat{x}_q)$ is a matrix of size 2×2 in 2D and 3×3 in 3D. The number of matrices $G = \{G(\hat{x}_0), G(\hat{x}_1), \dots\}$ are $n_{\text{cell}} \times n_q$. Notably, the number of matrices $G(\hat{x}_q)$ that need to be computed can be quite substantial, particularly in the case of 3D problems. For 3D problems, this translates to $n_{\text{cell}} \times n_q \times 3 \times 3$. However, $G(\hat{x}_q)$ is symmetric, hence, it is possible

to store only the lower or upper triangular matrix. For 3D problems, instead of storing 9 floating-point numbers, only 6 are stored.

Equations (5.8) and (5.9) illustrates the geometric data that can be precomputed and stored at the start of the simulation. The C++ implementation to compute $C(\hat{x}_q)$ and $G(\hat{x}_q)$ can be obtained from the FEniCSx-FUS repository. In section 5.5, the speed-up achieves from the precomputation of the geometric data is shown.

5.3 Sum-factorisation algorithm and implementation

A justification for using tensor-product elements is the benefit from the sum-factorisation algorithm. For quadrilateral and hexahedral elements, sum-factorisation provides computational speed-up by reducing the number of computation of the numerical quadrature evaluation [119, 114, 28]. Fundamentally, the sum-factorisation algorithm exploits the tensor-product structure of these elements to reduce the number of floating-point operations. In this section, the sum-factorisation algorithm and its implementation are outlined.

5.3.1 Standard method

As a point of reference, the cost of computation of the local stiffness operator based on the standard method is presented. To this end, consider the computation of equation (5.9) such that

$$b_j^k = \sum_{q=0}^{N_q} \left(\sum_{i=0}^{N_p} u_i \nabla_{\hat{x}} \phi_i^k(\hat{x}_q) \right) G(\hat{x}_q) \nabla_{\hat{x}} \phi_j^k(\hat{x}_q). \quad (5.10)$$

Expanding the above equation for each j 's yields

$$\begin{aligned} b_0^k &= (u_0 \nabla_{\hat{x}} \phi_0(\hat{x}_0) + u_1 \nabla_{\hat{x}} \phi_1(\hat{x}_0) + \cdots u_{N_p} \nabla_{\hat{x}} \phi_{N_p}(\hat{x}_0)) G(\hat{x}_0) \nabla_{\hat{x}} \phi_0^k(\hat{x}_0) \\ &\quad + (u_0 \nabla_{\hat{x}} \phi_0(\hat{x}_1) + u_1 \nabla_{\hat{x}} \phi_1(\hat{x}_1) + \cdots u_{N_p} \nabla_{\hat{x}} \phi_{N_p}(\hat{x}_1)) G(\hat{x}_1) \nabla_{\hat{x}} \phi_0^k(\hat{x}_1) \\ &\quad \dots \\ &\quad + (u_0 \nabla_{\hat{x}} \phi_0(\hat{x}_{N_q}) + u_1 \nabla_{\hat{x}} \phi_1(\hat{x}_{N_q}) + \cdots u_{N_p} \nabla_{\hat{x}} \phi_{N_p}(\hat{x}_{N_q})) G(\hat{x}_{N_q}) \nabla_{\hat{x}} \phi_0^k(\hat{x}_{N_q}) \end{aligned} \quad (5.11)$$

$$\begin{aligned} b_1^k &= (u_0 \nabla_{\hat{x}} \phi_0(\hat{x}_0) + u_1 \nabla_{\hat{x}} \phi_1(\hat{x}_0) + \cdots u_{N_p} \nabla_{\hat{x}} \phi_{N_p}(\hat{x}_0)) G(\hat{x}_0) \nabla_{\hat{x}} \phi_1^k(\hat{x}_0) \\ &\quad + (u_0 \nabla_{\hat{x}} \phi_0(\hat{x}_1) + u_1 \nabla_{\hat{x}} \phi_1(\hat{x}_1) + \cdots u_{N_p} \nabla_{\hat{x}} \phi_{N_p}(\hat{x}_1)) G(\hat{x}_1) \nabla_{\hat{x}} \phi_1^k(\hat{x}_1) \\ &\quad \dots \\ &\quad + (u_0 \nabla_{\hat{x}} \phi_0(\hat{x}_{N_q}) + u_1 \nabla_{\hat{x}} \phi_1(\hat{x}_{N_q}) + \cdots u_{N_p} \nabla_{\hat{x}} \phi_{N_p}(\hat{x}_{N_q})) G(\hat{x}_{N_q}) \nabla_{\hat{x}} \phi_1^k(\hat{x}_{N_q}) \end{aligned} \quad (5.12)$$

$\vdots$

$$\begin{aligned}
b_{N_p}^k &= (u_0 \nabla_{\hat{x}} \phi_0(\hat{x}_0) + u_1 \nabla_{\hat{x}} \phi_1(\hat{x}_0) + \cdots u_{N_p} \nabla_{\hat{x}} \phi_{N_p}(\hat{x}_0)) G(\hat{x}_0) \nabla_{\hat{x}} \phi_{N_p}^k(\hat{x}_0) \\
&+ (u_0 \nabla_{\hat{x}} \phi_0(\hat{x}_1) + u_1 \nabla_{\hat{x}} \phi_1(\hat{x}_1) + \cdots u_{N_p} \nabla_{\hat{x}} \phi_{N_p}(\hat{x}_1)) G(\hat{x}_1) \nabla_{\hat{x}} \phi_{N_p}^k(\hat{x}_1) \\
&\dots \\
&+ (u_0 \nabla_{\hat{x}} \phi_0(\hat{x}_{N_q}) + u_1 \nabla_{\hat{x}} \phi_1(\hat{x}_{N_q}) + \cdots u_{N_p} \nabla_{\hat{x}} \phi_{N_p}(\hat{x}_{N_q})) G(\hat{x}_{N_q}) \nabla_{\hat{x}} \phi_{N_p}^k(\hat{x}_{N_q})
\end{aligned} \tag{5.13}$$

for $j = 0, 1, \dots, N_p$. Notice that the expressions in blue are the same for each j 's. Consequently, the quadrature points can be looped through, and the contribution of each basis function is subsequently summed.

```

template <typename T, int Np, int Nq>
inline void transform3(T* __restrict__ G, T* __restrict__ dphi,
                      T __restrict__ coeff, T* __restrict__ in,
                      T* __restrict__ out)
{
    for (int iq = 0; iq < Nq; ++iq) {

        T w0 = 0.0;
        T w1 = 0.0;
        T w2 = 0.0;

        for (int id = 0; id < Np; ++id) {
            w0 += in[id] * dphi[iq * Np + id]; // dx
            w1 += in[id] * dphi[Np * Nq + iq * Np + id]; // dy
            w2 += in[id] * dphi[2 * Np * Nq + iq * Np + id]; // dz
        }

        const T _G = G + iq * 6;
        const T fw0 = coeff * (_G[0] * w0 + _G[1] * w1 + _G[2] * w2);
        const T fw1 = coeff * (_G[1] * w0 + _G[3] * w1 + _G[4] * w2);
        const T fw2 = coeff * (_G[2] * w0 + _G[4] * w1 + _G[5] * w2);

        for (int i = 0; i < Np; ++i) {
            out[i] += fw0 * dphi[iq * Np + i] + fw1 * dphi[Np * Nq + iq * Np + i] +
                    fw2 * dphi[2 * Np * Nq + iq * Np + i];
        }
    }
}

```

Listing 5.1 Standard implementation for the local stiffness operator.

Listing 5.1 presents the code that illustrates the standard implementation of the computation in equation (5.10). Note that this implementation includes the precomputation of geometric data that is stored in the variable `G`. The variable `in` denotes the local degrees-of-freedom u_i and the variable `dphi` denotes the gradient of the basis functions. In particular, the gradient of the basis functions is stored in a multidimensional array consisting of the gradient in each coordinate direction. Indeed,

the computational cost of the implementation in listing 5.1 is $N_q \times (N_p + N_p) = 2N_q N_p := 2(P+1)^6$ where P is the polynomial order of the basis function.

5.3.2 Sum-factorisation algorithm

The sum-factorisation algorithm to compute equation (5.10) is now presented. Sum-factorisation exploits the tensor-product structure of elements such as the quadrilateral and hexahedral elements. To illustrate, consider the hexahedral element where the basis function can be defined as a tensor product of the 1-dimensional basis function such that

$$\phi_i(\hat{x}) = \psi_{i_0}(\hat{x}_0)\psi_{i_1}(\hat{x}_1)\psi_{i_2}(\hat{x}_2) \quad (5.14)$$

where the subscript $(0, 1, 2)$ denotes the coordinate directions. Note that the superscript k has been removed for clarity. Substituting equation (5.14) into equation (5.10) yields

$$b_{j_0 j_1 j_2} = \sum_{q_0}^{N_Q} \sum_{q_1}^{N_Q} \sum_{q_2}^{N_Q} \left(\sum_{i_0}^{N_P} \sum_{i_1}^{N_P} \sum_{i_2}^{N_P} u_{i_0 i_1 i_2} \nabla_{\hat{x}} (\psi_{i_0}(\hat{x}_{q_0}) \psi_{i_1}(\hat{x}_{q_1}) \psi_{i_2}(\hat{x}_{q_2})) \right) \\ \times G(\hat{x}_{q_0 q_1 q_2}) \nabla_{\hat{x}} (\psi_{j_0}(\hat{x}_{q_0}) \psi_{j_1}(\hat{x}_{q_1}) \psi_{j_2}(\hat{x}_{q_2})), \quad (5.15)$$

where N_Q and N_P are the quadrature points and basis interpolation points in 1-dimension. In the case of the mass lumped finite element method, $N_Q = N_P = P + 1$, where P is the polynomial order of the basis function. A key element in the sum-factorisation algorithm is the tensor contraction operation. In mathematical notation, a tensor contraction operation between a 3D tensor A_{ijk} and a 2D tensor B_{kl} along the index k is defined as

$$C_{ijl} = \sum_k A_{ijk} B_{kl} \quad (5.16)$$

or more succinctly, using the Einstein summation notation

$$C_{ijl} = A_{ijk} B_{kl} \quad (5.17)$$

where the sum along the repeated index k is implied.

Without loss of generality, consider the $\hat{x}_0$ -coordinate direction. In this direction, the sum in the equation above reduces to

$$b_{j_0 j_1 j_2}^0 = \sum_{q_0}^{N_Q} \sum_{q_1}^{N_Q} \sum_{q_2}^{N_Q} \left(\sum_{i_0}^{N_P} \sum_{i_1}^{N_P} \sum_{i_2}^{N_P} u_{i_0 i_1 i_2} \nabla_{\hat{x}_0} (\psi_{i_0}(\hat{x}_{q_0})) \psi_{i_1}(\hat{x}_{q_1}) \psi_{i_2}(\hat{x}_{q_2}) \right) \\ \times G(\hat{x}_{q_0 q_1 q_2}) \nabla_{\hat{x}_0} (\psi_{j_0}(\hat{x}_{q_0})) \psi_{j_1}(\hat{x}_{q_1}) \psi_{j_2}(\hat{x}_{q_2}). \quad (5.18)$$

where $\nabla_{\hat{x}_0}$ is the gradient in the $\hat{x}_0$ direction. The key to the reduction in floating-point operation is by storing temporary variables once a sum in a single subscript is performed. In particular, summing the

equation above over i_2 gives

$$b_{j_0 j_1 j_2}^0 = \sum_{q_0}^{N_Q} \sum_{q_1}^{N_Q} \sum_{q_2}^{N_Q} \left(\sum_{i_0}^{N_P} \sum_{i_1}^{N_P} T_{i_0 i_1 q_2}^0 \nabla_{\hat{x}_0} (\psi_{i_0}(\hat{x}_{q_0})) \psi_{i_1}(\hat{x}_{q_1}) \right) \times G(\hat{x}_{q_0 q_1 q_2}) \nabla_{\hat{x}_0} (\psi_{j_0}(\hat{x}_{q_0})) \psi_{j_1}(\hat{x}_{q_1}) \psi_{j_2}(\hat{x}_{q_2}) \quad (5.19)$$

where $T_{i_0 i_1 q_2}^0 = \sum_{i_2}^{N_P} u_{i_0 i_1 i_2} \psi_{i_2}(\hat{x}_{q_2})$. Repeating the process for i_1 yields

$$b_{j_0 j_1 j_2}^0 = \sum_{q_0}^{N_Q} \sum_{q_1}^{N_Q} \sum_{q_2}^{N_Q} \left(\sum_{i_0}^{N_P} T_{i_0 q_1 q_2}^1 \nabla_{\hat{x}_0} (\psi_{i_0}(\hat{x}_{q_0})) \right) G(\hat{x}_{q_0 q_1 q_2}) \nabla_{\hat{x}_0} (\psi_{j_0}(\hat{x}_{q_0})) \psi_{j_1}(\hat{x}_{q_1}) \psi_{j_2}(\hat{x}_{q_2}) \quad (5.20)$$

where $T_{i_0 q_1 q_2}^1 = \sum_{i_1}^{N_P} T_{i_0 i_1 q_2}^0 \psi_{i_1}(\hat{x}_{q_1})$. Repeating the process for i_0 gives

$$b_{j_0 j_1 j_2}^0 = \sum_{q_0}^{N_Q} \sum_{q_1}^{N_Q} \sum_{q_2}^{N_Q} T_{q_0 q_1 q_2}^2 G(\hat{x}_{q_0 q_1 q_2}) \nabla_{\hat{x}_0} (\psi_{j_0}(\hat{x}_{q_0})) \psi_{j_1}(\hat{x}_{q_1}) \psi_{j_2}(\hat{x}_{q_2}) \quad (5.21)$$

where $T_{q_0 q_1 q_2}^2 = \sum_{i_0}^{N_P} T_{i_0 q_1 q_2}^1 \nabla_{\hat{x}_0} \psi_{i_0}(\hat{x}_{q_0})$. Summing over the quadrature points starting with q_2 yields

$$b_{j_0 j_1 j_2}^0 = \sum_{q_0}^{N_Q} \sum_{q_1}^{N_Q} T_{q_0 q_1 j_2}^3 G(\hat{x}_{q_0 q_1}) \nabla_{\hat{x}_0} (\psi_{j_0}(\hat{x}_{q_0})) \psi_{j_1}(\hat{x}_{q_1}) \quad (5.22)$$

where $T_{q_0 q_1 j_2}^3 = \sum_{q_2}^{N_Q} T_{q_0 q_1 q_2}^2 G(\hat{x}_{q_0 q_1 q_2}) \psi_{j_2}(\hat{x}_{q_2})$. Repeating the process gives

$$b_{j_0 j_1 j_2}^0 = \sum_{q_0}^{N_Q} T_{q_0 j_1 j_2}^4 G(\hat{x}_{q_0}) \nabla_{\hat{x}_0} (\psi_{j_0}(\hat{x}_{q_0})) \quad (5.23)$$

where $T_{q_0 j_1 j_2}^4 = \sum_{q_1}^{N_Q} T_{q_0 q_1 j_2}^3 G(\hat{x}_{q_0 q_1}) \psi_{j_1}(\hat{x}_{q_1})$. Finally, the sum over q_0 is performed. The process for $\hat{x}_1$ and $\hat{x}_2$ are the same. The computational cost of the above steps is then

$$N_P^3 N_Q + N_P^2 N_Q^2 + N_P N_Q^3 + N_P N_Q^3 + N_Q^2 N_P^2 + N_P^3 N_Q. \quad (5.24)$$

If $N_D = N_Q = P + 1$, then the computational cost of the sum factorisation method is of order $\mathcal{O}(P^4)$ while the standard approach is order $\mathcal{O}(P^6)$. Figure 5.1 shows the comparison in timings between the standard algorithm with the sum-factorisation algorithm. Notably, the sum-factorisation algorithm is clearly faster for higher order basis function. A further advantage of the sum-factorisation algorithm is the tabulation of the basis function is significantly smaller in contrast to the standard method since only the 1D basis needs to be tabulated, hence, reduces the memory requirement of the solver.

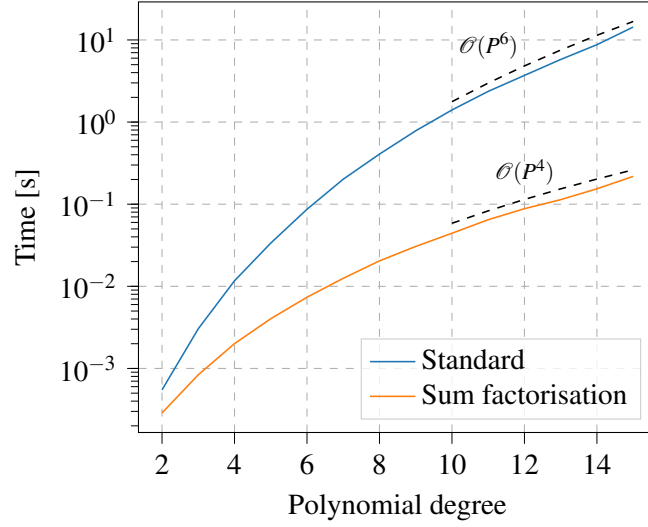


Fig. 5.1 The comparison between the theoretical computational cost of the standard and sum-factorisation algorithm for the assembly of finite element vector. Notice that as the polynomial order increases, the cost difference between the two algorithms becomes more evident. The dashed lines represent the gradients of the polynomial order.

5.3.3 Tensor contraction through matrix-matrix multiplication

While the sum-factorisation algorithm shown previously provides an illustration of the steps involve in the algorithm, in practice, the tensor contraction—the summation along an index—process is typically performed using matrix-matrix multiplication. This approach is favourable because compilers can perform auto-vectorisation—computing multiple floating-point numbers at once. As a result, further speed-up can be achieved. However, this feature is highly dependent upon the compiler. Nonetheless, FEniCSx-FUS always seeks to compile with auto-vectorisation.

To illustrate the sum-factorisation algorithm using matrix-matrix multiplication, the variables in equation (5.18) are viewed as tensors. For example, the evaluation of basis function at quadrature points such as $\psi_{i_2}(\hat{x}_2)$ can be viewed as a matrix of size $N_Q \times N_P$ while the degrees-of-freedom $u_{i_0 i_1 i_2}$ is a tensor of size $N_P \times N_P \times N_P$. Accordingly, rewriting equation (5.18) such that

$$b_{j_0 j_1 j_2}^0 = U_{i_0 i_1 i_2} \Psi_{i_2 q_2} \Psi_{i_1 q_1} \nabla_{\hat{x}_0} \Psi_{i_0 q_0} G_{q_0 q_1 q_2} \Psi_{j_2 q_2} \Psi_{j_1 q_1} \nabla_{\hat{x}_0} \Psi_{j_0 q_0} \quad (5.25)$$

where Ψ is the matrix of the evaluation of the basis function at quadrature points. The symbol $\nabla_{\hat{x}_0} \Psi$ denotes the matrix of the evaluation of the gradient of the basis functions at quadrature points. Both of these matrices are tabulated by the Basix library [140]. Performing tensor contraction on i_2 yields

$$b_{j_0 j_1 j_2}^0 = T_{i_0 i_1 q_2} \Psi_{i_1 q_1} \nabla_{\hat{x}_0} \Psi_{i_0 q_0} G_{q_0 q_1 q_2} \Psi_{j_2 q_2} \Psi_{j_1 q_1} \nabla_{\hat{x}_0} \Psi_{j_0 q_0} \quad (5.26)$$

where $T_{i_0 i_1 q_2} = U_{i_0 i_1 i_2} \Psi_{i_2 q_2}$, in this instance, the tensor $U_{i_0 i_1 i_2}$ can be viewed as a matrix of $N_P \times N_P$ rows and N_P columns. The Einstein notation is used, which implies summation over the repeated

indices. Transposing the tensor yields

$$b_{j_0 j_1 j_2}^0 = T_{i_0 q_2 i_1} \Psi_{i_1 q_1} \nabla_{\hat{x}_0} \Psi_{i_0 q_0} G_{q_0 q_1 q_2} \Psi_{j_2 q_2} \Psi_{j_1 q_1} \nabla_{\hat{x}_0} \Psi_{j_0 q_0} \quad (5.27)$$

where $T_{i_0 q_2 i_1}$ is a matrix of $N_P \times N_Q$ rows and N_P columns. Repeating the process gives

$$b_{j_0 j_1 j_2}^0 = T_{i_0 q_2 q_1} \nabla_{\hat{x}_0} \Psi_{i_0 q_0} G_{q_0 q_1 q_2} \Psi_{j_2 q_2} \Psi_{j_1 q_1} \nabla_{\hat{x}_0} \Psi_{j_0 q_0} \quad (5.28)$$

where $T_{i_0 q_2 q_1} = T_{i_0 q_2 i_1} \Psi_{i_1 q_1}$ and transposing leads to

$$b_{j_0 j_1 j_2}^0 = T_{q_1 q_2 i_0} \nabla_{\hat{x}_0} \Psi_{i_0 q_0} G_{q_0 q_1 q_2} \Psi_{j_2 q_2} \Psi_{j_1 q_1} \nabla_{\hat{x}_0} \Psi_{j_0 q_0}. \quad (5.29)$$

Repeating the process of tensor contraction and transposition yields

$$b_{j_0 j_1 j_2}^0 = T_{q_0 q_1 q_2} \nabla_{\hat{x}_0} \Psi_{i_0 q_0} G_{q_0 q_1 q_2} \Psi_{j_2 q_2} \Psi_{j_1 q_1} \nabla_{\hat{x}_0} \Psi_{j_0 q_0}. \quad (5.30)$$

where $T_{q_0 q_1 q_2} = T_{q_1 q_2 i_0} \nabla_{\hat{x}_0} \Psi_{i_0 q_0} G_{q_0 q_1 q_2}$. The final result is obtained by repeating the tensor contraction process and transposition for index q 's. Listing 5.2 shows the code snippet that implements the sum-factorisation algorithm using matrix-matrix multiplication and matrix transposition.

```
T1.fill(0.0);
T2.fill(0.0);
T3.fill(0.0);
T4.fill(0.0);

// Apply contraction in the x-direction
fw0_.fill(0.0);
contract<T, N, N, N, N, N, true>(dphi_, x_.data(), fw0);

// Apply contraction in the y-direction
fw1_.fill(0.0);
transpose<T, N, N, N, N, N, N*N, 1>(x_.data(), T1.data());
contract<T, N, N, N, N, N, true>(dphi_, T1.data(), T2.data());
transpose<T, N, N, N, N, N, N*N, 1>(T2.data(), fw1);

// Apply contraction in the z-direction
fw2_.fill(0.0);
transpose<T, N, N, N, N, 1, N, N*N>(x_.data(), T3.data());
contract<T, N, N, N, N, N, true>(dphi_, T3.data(), T4.data());
transpose<T, N, N, N, N, 1, N, N*N>(T4.data(), fw2);

// Apply transform
T* G = G_.data() + c * Nd * 6;
stiffness::transform<T, P, Nd>(G, coeffs[c], fw0, fw1, fw2);

T1.fill(0.0);
```

```

T2.fill(0.0);
T3.fill(0.0);
T4.fill(0.0);

// Apply contraction in the x-direction
y0_.fill(0.0);
contract<T, N, N, N, N, false>(dphi_, fw0, y0_.data());

// Apply contraction in the y-direction
y1_.fill(0.0);
transpose<T, N, N, N, N, N*N, 1>(fw1, T1.data());
contract<T, N, N, N, N, false>(dphi_, T1.data(), T2.data());
transpose<T, N, N, N, N, N*N, 1>(T2.data(), y1_.data());

// Apply contraction in the z-direction
y2_.fill(0.0);
transpose<T, N, N, N, 1, N, N*N>(fw2, T3.data());
contract<T, N, N, N, N, false>(dphi_, T3.data(), T4.data());
transpose<T, N, N, N, 1, N, N*N>(T4.data(), y2_.data());

for (std::int32_t i = 0; i < Nd; ++i)
    y_array[tensor_dofmap_[c * Nd + i]] += y0_[i] + y1_[i] + y2_[i];

```

Listing 5.2 Sum-factorisation implementation for the local stiffness operator. Note that some steps are omitted in the apply contraction operation, this is because the basis function evaluation matrix is an identity matrix if the quadrature points are used as the interpolation points for the basis function. As a result, the tensor contraction is equivalent to multiplying with an identity matrix, which returns the same matrix, thus, can be omitted.

Mathematically, the tensor contraction operation can be written as a matrix-matrix multiplication such that

$$C_{ad} = A_{ak} B_{kd} \quad (5.31)$$

where A is a 2D tensor of size $N_a \times N_k$, and B and C are 3D tensors of size $N_k \times N_b \times N_c$. Hence, $N_d = N_b \times N_c$. Listing 5.3 shows the tensor contraction implementation of equation (5.31), the main engine behind the efficiency of the sum-factorisation algorithm. It is highlighted here that the implementation takes into account the memory access pattern and data locality to better utilise the architecture memory bandwidth. This is evident whereby the innermost loop is iterated using a row-major layout—the data layout used by the C++ language. In section 5.5, it will be shown that the implementation achieves a good fraction of peak performance.

```

/// Compute the tensor contraction C[a,b,c] <- A[a, k] B[k, b, c]
/// as a matrix-matrix multiplication C[a,{b, c}] = A[a, k] B[k, {b, c}]
/// K is the contraction index
template <typename T, int Nk, int Na, int Nb, int Nc, bool transpose>
static inline void contract(const T* __restrict__ A, const T* __restrict__ B,
                           T* __restrict__ C) {
    // d = {b, c} is a multi-index d = b * Nc + c

```

```
constexpr int Nd = Nb * Nc;

if constexpr (transpose) {
    for (int k = 0; k < Nk; k++)
        for (int a = 0; a < Na; a++)
            for (int d = 0; d < Nd; d++)
                C[a * Nd + d] += A[a * Nk + k] * B[k * Nd + d];
} else {
    for (int k = 0; k < Nk; k++)
        for (int a = 0; a < Na; a++)
            for (int d = 0; d < Nd; d++)
                C[a * Nd + d] += A[k * Na + a] * B[k * Nd + d];
}
}
```

Listing 5.3 The tensor contraction implementation. The tensor contraction is implemented as a matrix-matrix multiplication operation. In this regard, the input matrix A (the basis function evaluation) is multiply with the degrees-of-freedom tensor B that is *reshape* as a 2D matrix.

5.4 Reduced precision

To further enhance the solver efficiency, in addition to the 64-bit float implementation, a 32-bit float implementation is also provided. This section presents the implementation of reduced precision in the solver with particular emphasis on the transition of double (64-bit) to single (32-bit) precision float. The implementation only requires minimal changes to the code.

C++ templating allows the same code to be used with different data types. A code snippet of the C++ templating that is implemented in the solver is shown in listing 5.4. In the listing, it shows that the `WesterveltSpectral3D` takes two template parameters, namely the `typename` and `int` parameters. The parameter `T` represents the float type and the parameter `P` represents the order of the polynomial basis function. Accordingly, the implementation of the solver is kept concise by using C++ templates to avoid code repetition. The speed-up improvements due to reduced precision is shown in section 5.5.

```
template <typename T, int P>
class WesterveltSpectral3D {
public:
    WesterveltSpectral3D(...)
    {
        ...
    }
}
```

Listing 5.4 An example of C++ templating used in the solver. Note the `typename T` within the angle bracket in the first line, which enables the user the option to use either 32 or 64-bit floating-point number. To use 32-bit float, the user can pass the `float` data type while to use 64-bit float, the user can pass the `double` data type.

5.5 Performance analysis

In the preceding sections, three approaches to accelerate the FEniCSx-FUS solver performance were presented, namely the precomputation of geometric data, the sum-factorisation algorithm, and the reduced precision implementation. In this section, their performances are analysed based on the speed-up achieved. Also, to verify that the implementation is optimal, a comparison of the tensor contraction implementation is performed with an established tensor manipulation library. Further, the fraction of peak performance achieved by the sum-factorisation implementation is discussed. The strong scaling¹ result of the solver is also presented.

5.5.1 Precomputing geometric data

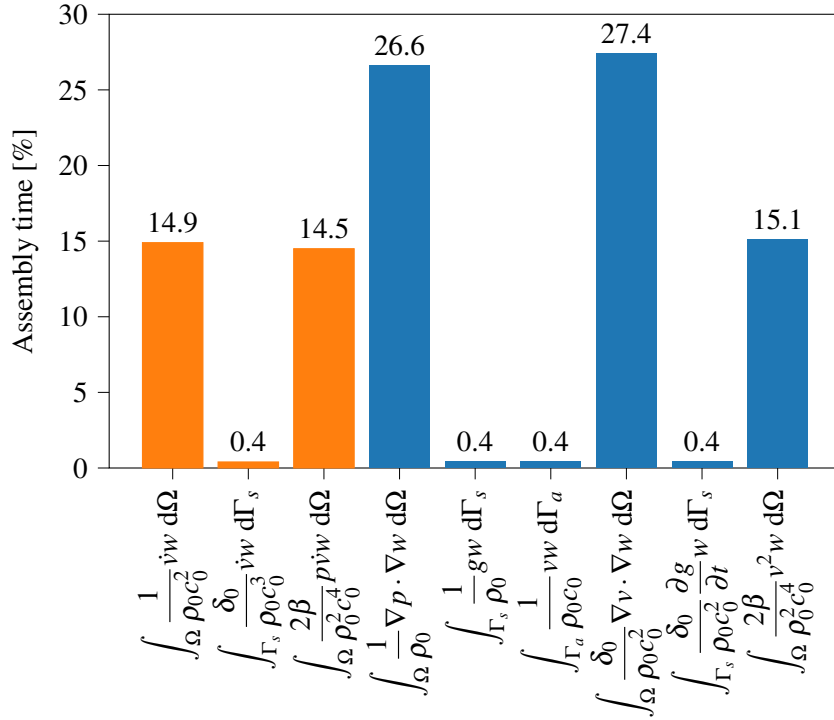


Fig. 5.2 The percentage of time spent for vector assembly with precomputation of geometric data. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using $P4$ Lagrange element. The total time for assembly is 0.55 seconds. The orange bars represent the terms on the left-hand side and the blue bars represent the terms on the right-hand side.

Figure 5.2 shows the percentage of time spent to assemble each term that appears in the forms definition of listing 4.1 with precomputing geometric data. It is observed that in contrast to the percentages in figure 4.8, the percentage of time spent to assemble the vectors of cell-wise gradient operators have increased. This is due to the reduction in assembly time for the mass operator term that

¹A test whereby the number of processors is increased while the problem size remains constant.

consequently effects the percentage of the assembly time for the stiffness operator terms. However, the total assembly time decreases from 0.76 seconds to 0.55 seconds.

Figure 5.3 shows the comparison between the assembly time of the FEniCSx implementation against the precomputing geometric data implementation. A single form was chosen to represent the cell-wise terms that appears in the form definition of listing 4.1. In particular, the $\int_{\Omega} c_m(x)uw \, d\Omega$ represents the mass operator terms (e.g., $\int_{\Omega} 1/(\rho_0 c_0^2) \dot{v}w \, d\Omega$) and $\int_{\Omega} c_s(x) \nabla u \cdot \nabla w \, d\Omega$ represents the stiffness operator terms (e.g., $\int_{\Omega} 1/\rho_0 \nabla p \cdot \nabla w \, d\Omega$). A 1.5 times speed-up of the cell-wise operator is observed for the uw term as compared to the FEniCSx implementation. Additionally, for the $\nabla u \cdot \nabla w$ term, a 1.2 times speed-up is observed. Even though the differences may seem minimal, remember that the assembly of the vectors are performed many times during simulation, on the scale of 10^4 to 10^5 time-steps, which can impact the total simulation time. The difference in speed-up between the two terms can be explained by the difference in the amount of data movement. In particular, equation (5.8) only requires a single floating-point number at each quadrature point in contrast to equation (5.9), which requires 6 floating-point number. Overall, the precomputation of geometric data demonstrates improvement in terms of the assembly time.

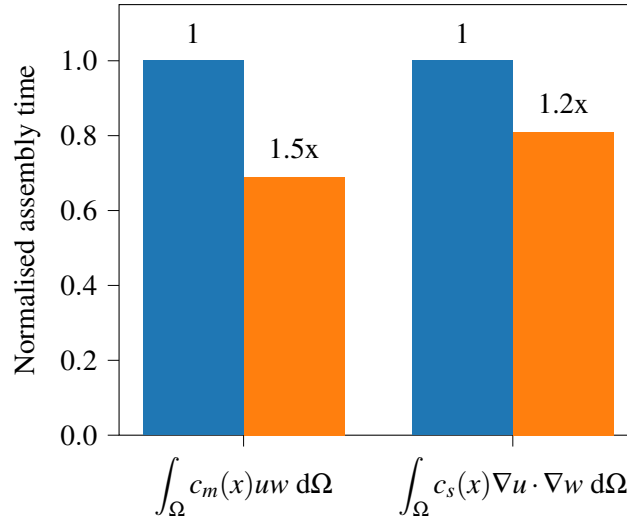


Fig. 5.3 A comparison between FEniCSx assembly and the assembly with precomputation of geometric data. This test was performed on the Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom using $P4$ Lagrange element. The blue bars represent the FEniCSx implementation and the orange bars represent the precomputation of geometric data implementation.

5.5.2 Sum-factorisation implementation

Figure 5.4 shows the assembly time of the cell-wise operator for 3 different implementation. A significant speed-up is observed when sum-factorisation is used. More precisely, for the uw term, an order of magnitude speed-up is achieved as compared to the FEniCSx implementation. Comparing with the PC implementation, the sum-factorisation implementation is around 7 times faster. For the

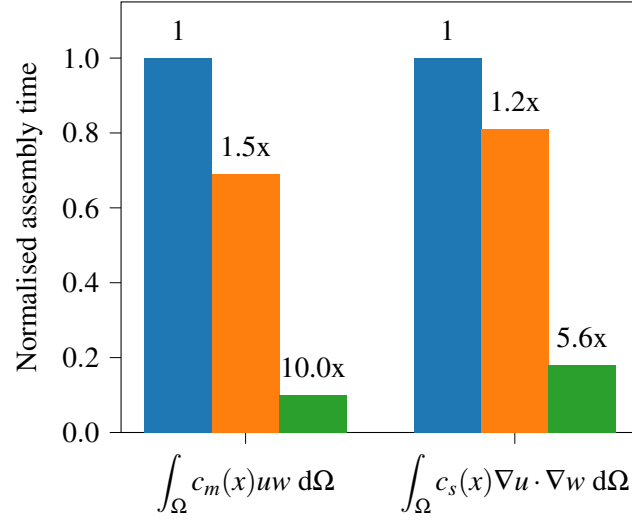


Fig. 5.4 A comparison between 3 different implementation, namely FEniCSx assembly, the precomputed geometric (PC) assembly, and the sum factorisation assembly. The **blue** bar represents FEniCSx implementation, the **orange** bar represents the precomputed geometric data implementation, and the **green** bar represents the sum-factorisation implementation. This test was performed using the Intel Icelake CPU on 76 processes with 60×10^6 degrees-of-freedom with $P4$ Lagrange element using double floating-point precision.

$\nabla u \cdot \nabla w$ term, a speed-up of 5.6 times is achieved as compared to the FEniCSx implementation. The speed-up can be explained by the well-designed implementation of the sum-factorisation algorithm. In particular, the speed-up can be attributed to the reduction in the size of the data and the efficient memory access pattern of the implementation.

Comparison with Fastor

A computational experiment was conducted to verify that the implementation of the sum-factorisation algorithm is performing competitively. To this end, the tensor contraction implementation is compared against the Fastor C++ library—a high performance tensor manipulation library [131].

The assembly time of the $\nabla u \cdot \nabla w$ term is compared. Figure 5.5 shows the normalised assembly time between the implementation and the Fastor implementation. It is observed that the implementation is quite competitive against the Fastor implementation, with the differences in assembly time being small. The small discrepancies can be explained by the use of Intel vector intrinsics in Fastor. This enables Fastor to fully leverage the performance of the Intel CPUs. Despite that, the Fastor implementation is restricted to Intel processor while the implementation in this work is more general.

Fraction of peak performance

To further test the performance of the implementation, the fraction of peak performance is measured using the *likwid* toolbox [164, 137]. The *likwid* toolbox provides several benchmarks to measure the performance of a code or architecture. In this study, the *likwid-bench* [165] application are used

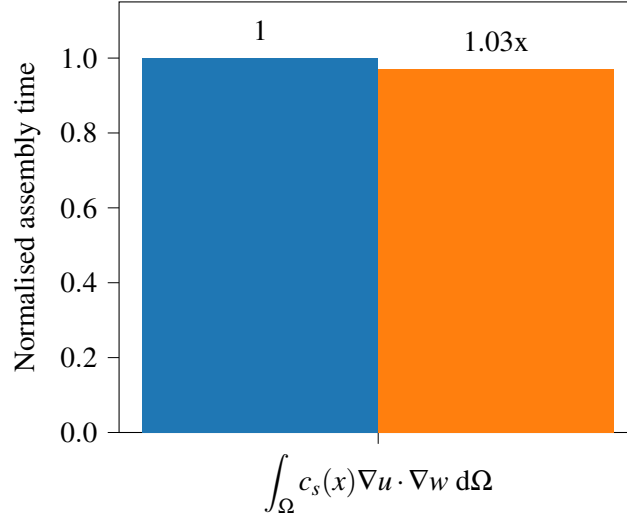


Fig. 5.5 A comparison between the vector assembly time using the implementation in this work against the Fastor implementation. The blue bar represents the implementation in this work and the orange bar represents Fastor implementation. This test was performed using the Intel Icelake CPU on 76 processes with 60×10^6 degrees-of-freedom with $P4$ Lagrange element using double floating-point precision.

to compare the performance of the assembly kernels with the benchmark. In particular, the triad benchmark is utilised, which performs the operation $A[i] = B[i] + C[i] * D[i]$, where A, B, C , and D are vectors. The benchmark was chosen as it is the closest operation that is comparable with the assembly kernel operations.

The performance of an implementation can be measured by the total size in bytes that can be loaded per time frame such that

$$\text{Performance} = \frac{\text{Number of bytes}}{\text{Time}}. \quad (5.32)$$

Thus, a vector assembly requires

$$\text{Number of bytes} = 3X + C + \text{DOFMAP} + G \quad (5.33)$$

where X is the global number of degrees-of-freedom, C is the global number of cells, DOFMAP is the degrees-of-freedom map, and G is the precomputed geometric data. The difference between the assembly of the mass operator and the stiffness operator are in the variable G , where the stiffness operator requires 6 loads as opposed to a single load for mass operator.

Figure 5.6 shows the performance of the assembly implementation. It is observed that the implementation achieves a good fraction of peak performance whereby the lowest performance achieves 72% of peak performance— $P3$ of the stiffness operator. A rise between $P3$ and $P4$ is observed, which can be attributed to the auto-vectorisation that is performed by the compiler. In terms of the mass operator assembly, a consistent and uniform pattern is observed across different polynomial order, it is because the mass operator assembly process is predominantly memory-bound, i.e., the

process involves significant data movement instead of computation. The result also demonstrates that the algorithm is correctly implemented since the mass operator assembly should be inexpensive computationally, i.e., the number of bytes that needs to be loaded is smaller than for the stiffness operator assembly.

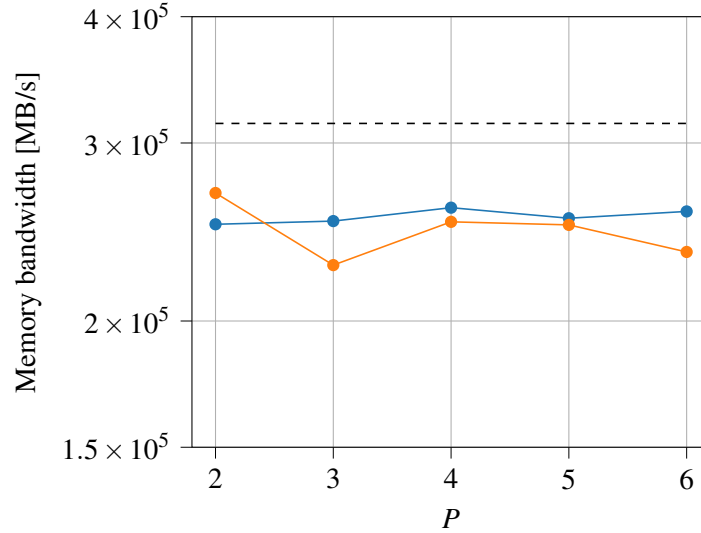


Fig. 5.6 Performance of the vector assembly of the mass operator and the stiffness operator for different polynomial order. The test was conducted on double precision floating-point number using Intel Icelake CPU with 76 processes. The dashed line represents the peak performance of the architecture for triad benchmark. The blue line and orange line represent the performance achieved by the assembly implementation. Blue represents the mass operator and orange represents the stiffness operator.

5.5.3 Reduced precision

Figure 5.7 shows the comparison of the assembly time between the double and single precision floats implementation. Indeed, the speed-up achieved by using single precision floating-point numbers is near perfect, i.e., close to 2 times faster. In terms of the fraction of peak performance, a consistent pattern between the single precision performance and the double precision performance is observed—refer to figures 5.6 and 5.8. Thus, verifies that the implementation of the single precision float is also optimal.

5.5.4 Strong scaling

Figure 5.9 shows the result of a strong scaling experiment on up to 28 compute nodes of Intel Icelake CPU on the CSD3 machine at the University of Cambridge. The largest computation was performed using 2044 MPI ranks. The results demonstrate an almost perfect scaling between 0.01 seconds per time step up to 0.4 seconds per time step. A small deviation from perfect scaling is observed for 126 million degrees-of-freedom at 2044 number of cores, which corresponds to around 60000 degrees-of-freedom per core. However, this deviation is minimal since the solver still maintains a

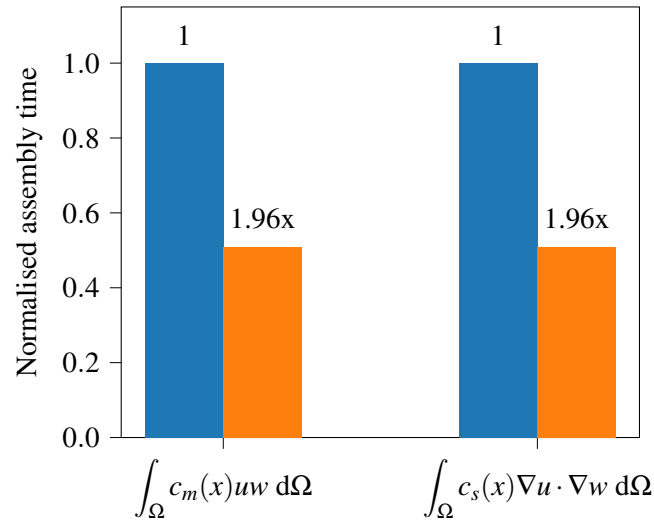


Fig. 5.7 A comparison between the vector assembly time using double and single precision floating-point numbers. The blue bar represents double precision and the orange bar represents single precision. This test was performed on Intel Icelake CPU using 76 processes with 60×10^6 degrees-of-freedom.

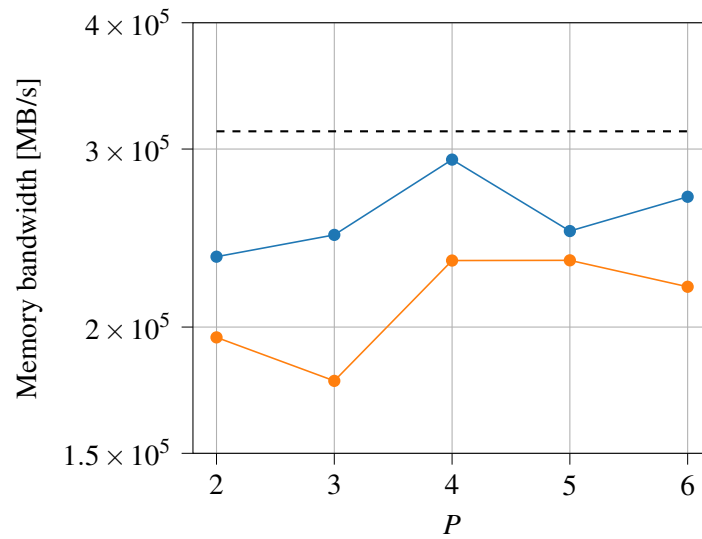


Fig. 5.8 Performance of the mass operator and the stiffness operator for different polynomial orders. The test was conducted using single precision floating-point numbers on Intel Icelake CPU with 76 processes. The dashed line represents the peak performance of the architecture for triad benchmark. The blue line and orange line represent the performance achieved by the assembly implementation. Blue represents the mass operator and orange represents the stiffness operator.

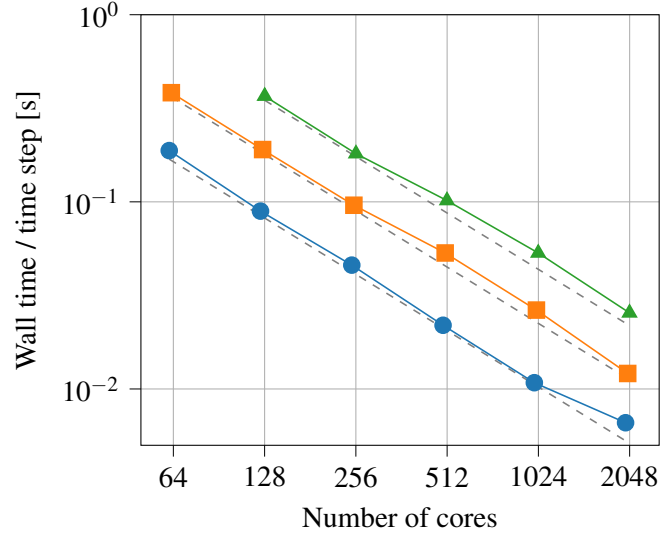


Fig. 5.9 Strong scaling analysis of the FEniCSx-FUS solver on Intel Icelake CPU for problem sizes of 31 million (blue), 62 million (orange), and 126 million (green) degrees-of-freedom, respectively. The dashed line represents perfect strong scaling. The computational experiment was performed using between 2305 and 3697 time steps for 4th order explicit Runge–Kutta method. The data shows the run time per step.

parallel efficiency of 90%. Accordingly, the result verifies the excellent performance of the MPI implementation.

5.6 Summary

In this chapter, techniques to accelerate the finite element integration computation for the FEniCSx-FUS solver were implemented. Since the mesh for a focused ultrasound simulation is typically fixed over the entire simulation, the geometry data was precomputed. The sum-factorisation algorithm was also implemented. The algorithm reduces the number of floating-point operations in the computation of the finite element integrals. Also, reduced precision was implemented to leverage 32-bit floats. These implementations were validated to ensure optimal performance. It was found that each of the implementations speed-up the assembly time of the cell-wise finite element integrals. Additionally, it was demonstrated that the implementation is competitive with an established library and performed at a good fraction of peak performance on a high-performance computing architecture. The FEniCSx-FUS solver also demonstrates good strong scaling performance.

Chapter 6

Validation of the FEniCSx-FUS solver

In this chapter, the validation of FEniCSx-FUS is presented via several numerical experiments—compared to published results. Sections 6.1 and 6.2 use the linear wave model and section 6.3 uses the Westervelt model.

In section 6.1, the importance of the windowing function on the source boundary condition is illustrated. In section 6.2, the validation of the solver for transcranial focused ultrasound applications is presented. The benchmark problems presented in [16] are considered. In this regard, the outputs of FEniCSx-FUS are compared with k-Wave [160], an established solver widely used for focused ultrasound simulation. Finally, section 6.3 presents the validation for high-intensity focused ultrasound applications. In particular, comparisons of FEniCSx-FUS with two solvers used for HIFU simulation are presented, namely the VINES [66] and HITU simulator [147].

6.1 Source windowing

Consider wave field propagation in a homogeneous medium with no attenuation. At $t = 0$, when the source is *switched on*. If the transition is applied too sharply (or even discontinuously), unwanted spurious waves will be generated that will pollute the numerical solution. To mitigate this, it is desirable to switch on the source function smoothly, which can be achieved by applying a windowing function with a desired degree of smoothness. There are several choices available, such as the Gaussian window [144], the Planck-tapered window [1], and the Tukey window [25]. In this work, the Tukey window is utilised since the continuity of the Tukey window (C^1) is sufficient to handle the source function.

The Tukey window is defined as

$$w(t) = \begin{cases} \frac{1}{2} \left(1 - \cos \left(\frac{\pi f_0 t}{\alpha} \right) \right) & 0 \leq t < \frac{\alpha}{f_0} \\ 1 & t \geq \frac{\alpha}{f_0} \end{cases} \quad (6.1)$$

where α is the window length. Multiplying the above equation with the source function yields

$$g(t) = w(t)\rho_0\omega_0c_0\cos(\omega_0t). \quad (6.2)$$

From the forms definition of equation (4.1), the term $\partial g/\partial t$ is also required, using the chain rule on equation (6.2) gives

$$\frac{\partial g}{\partial t} = -w(t)\rho_0\omega_0^2c_0\sin(\omega_0t) + w'(t)\rho_0\omega_0c_0\cos(\omega_0t). \quad (6.3)$$

Figure 6.1 shows a comparison between a source function without windowing and with windowing, with the x -axis representing the time. Observed that there is a gradual change with respect to time for the source function when windowing is applied, which is expected to provide a smooth transition when the source is *switched on*, thus reducing the generation of spurious waves.

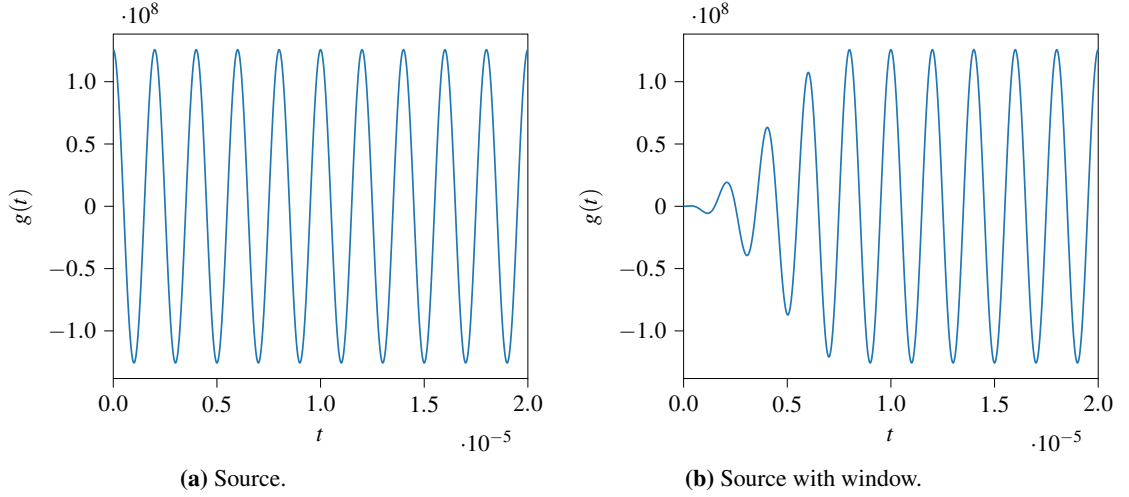


Fig. 6.1 Source function without and with windowing.

To verify that windowing does reduce the generation of spurious waves, consider a simple linear plane wave propagation problem, as shown in figure 6.2. The plane wave propagation problem is modelled using the linear wave equation with no attenuation ($\delta = 0$). The left-hand wall is set to be the source boundary condition of equation (2.32d) and the right-hand wall is set to be the absorbing boundary condition of equation (2.32c). The top and bottom wall is set as a zero Neumann boundary condition, which ensures that the waves propagate in one direction, i.e., to the right of the domain.

Figure 6.3 shows the rate of convergence of the finite element solution for plane wave problem shown in figure 6.2 at $t = L/c_0 + 6.0/f_0$, where L is the domain length (i.e., length of Γ_3), c_0 is the medium speed of sound, and f_0 is the source frequency. These parameters ensure that the plane wave has fully propagated to the right-hand wall. In figure 6.3a, it can be seen that the convergence of the finite element solution is not as expected, i.e. $\mathcal{O}(h^{P+1})$ rate of convergence, which is due to the generation of spurious waves when the source is *switched on* at $t = 0$. However, observed that in

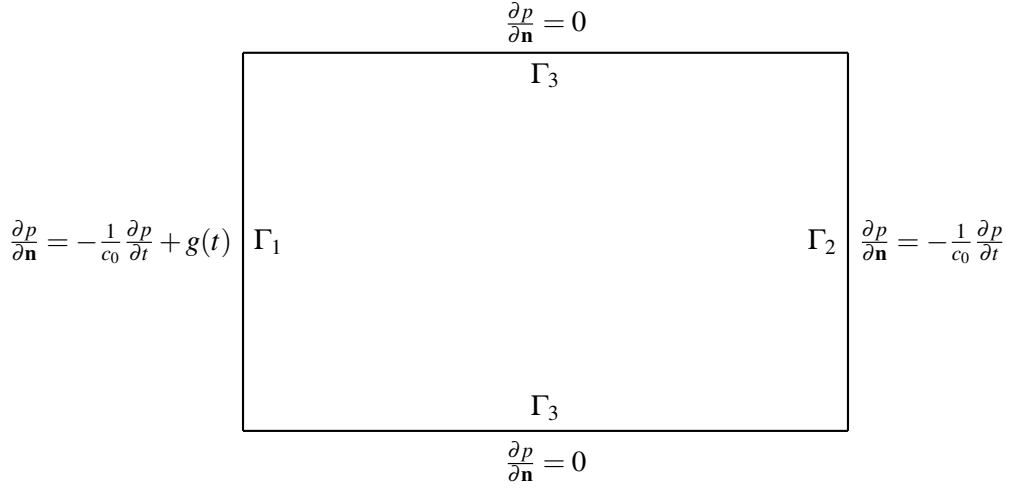


Fig. 6.2 Oscillating wall. The source boundary condition and absorbing boundary condition are prescribed on Γ_1 and Γ_2 , respectively. On Γ_3 , a zero Neumann boundary condition is prescribed. This setup ensures that the plane wave propagates to the right and in a single direction.

figure 6.3b, the convergence of the finite element solution behaves as expected, thus verifying that source windowing does, in fact, removes the generation of spurious waves. This result highlights the importance of the windowing function on the source boundary condition. In FEniCSx-FUS, the windowing function is integrated with the source boundary condition. Accordingly, in subsequent sections, the source modelling of the FEniCSx-FUS solver includes the windowing function.

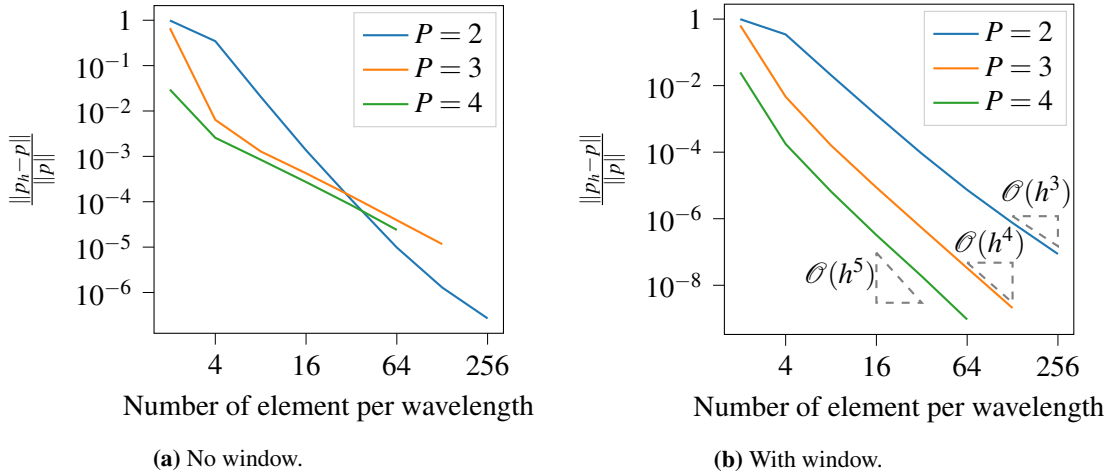


Fig. 6.3 The rate of convergence of the finite element solution for the plane wave propagation problem. The symbols p and p_h denote the analytical solution and the finite element solution of the pressure field, respectively.

6.2 Transcranial focused ultrasound

FEniCSx-FUS solutions are validated against several transcranial focused ultrasound (tFUS) problems used in [16]. In particular, seven different benchmark problems of increasing complexity are considered. The benchmark problems include a single-planar bone immersed in water, a multi-layer bone, and a partition of a human skull. The benchmarks also consider two different transducer types (a focused bowl transducer and a plane piston transducer, both operating at 500 kHz). These give a total of 14 different numerical experiments. Several parameters such as the position, size, and magnitude of the focal region are compared with a reference solver, namely the k-Wave solver [160].

6.2.1 Model equation

The model equation for the benchmark problems is the linear wave equation with attenuation that reads

$$\frac{1}{\rho_0} \nabla^2 p - \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} + \frac{\delta}{\rho_0 c_0^2} \nabla^2 \frac{\partial p}{\partial t} = 0 \quad (6.4)$$

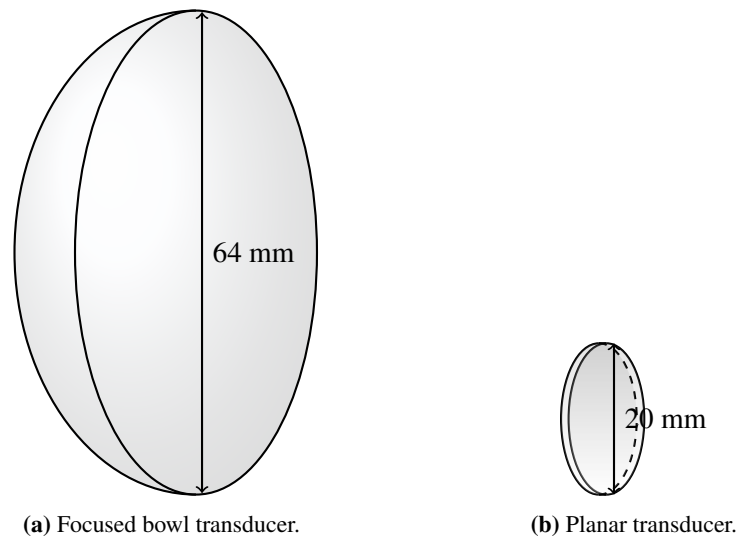
where p is the pressure field. The medium parameters c_0 , ρ_0 , and δ denote the speed of sound, density, and diffusivity of sound, respectively. The medium parameters can be spatially varying. Note that equation (6.4) can be obtained by setting $\beta = 0$ in the Westervelt model of equation (2.27).

6.2.2 Transducer configurations

The benchmark problems consider two different transducer types, namely a focused bowl transducer and a circular plane piston transducer. The focused bowl transducer is a spherical bowl transducer with a focal length of 64 mm and an aperture diameter of 64 mm, as shown in figure 6.4a. It represents a single-element transducer that is widely used in tFUS applications [187]. The plane piston transducer is a circular transducer with a diameter of 20 mm, as illustrated in figure 6.4b. This type of transducer typically represents multi-array elements. The diameter of 20 mm is chosen to obtain identifiable beam characteristics in the simulation [16]. In practice, the diameter of a piston transducer is typically smaller. Both transducers were set to run at 500 kHz with a constant surface speed of 0.04 m s^{-1} .

6.2.3 Material parameters

The values for the material parameters used in the simulations are given in table 6.1. These values are obtained from a range of literature on biological tissues [178, 180, 128, 124, 38, 112, 59]. It should be noted that this benchmark does not include nonlinearity, hence, the coefficient of nonlinearity $\beta = 0$. In accordance with [16], the attenuation power law of each material follows the frequency squared dependence.

**Fig. 6.4** Transducers configuration.**Table 6.1** Material parameters used in the simulations.

	c (m s ⁻¹)	ρ (kg m ⁻³)	α (dB cm ⁻¹ at 500 kHz)
Water	1500	1000	0
Skin	1610	1090	0.2
Brain	1560	1040	0.3
Cortical bone	2800	1850	4
Trabecular bone	2300	1700	8

6.2.4 Simulation setup

The benchmarks' meshes were generated using the open-source software Gmsh [62] and are available in this repository. Examples of the meshes used in the FEniCSx-FUS simulation are shown in figure 6.5. A 4th polynomial degree element with 2–3 elements per wavelength was used in all simulations. This was found to be sufficient for all benchmark problems. The temporal step size is chosen based on [149]

$$\Delta t = \text{CFL} \frac{h}{c_0 P^2}, \quad (6.5)$$

where CFL denotes the CFL number, h denotes the minimum mesh size, c_0 denotes the fastest speed of sound between the mediums, and P denotes the order of the Lagrange basis function, where $P = 4$. The CFL condition is necessary to ensure the stability of the numerical solutions. For the benchmark problems considered, it was found that the FEniCSx-FUS solver is stable for CFL condition in the range of 0.1 to 0.7.

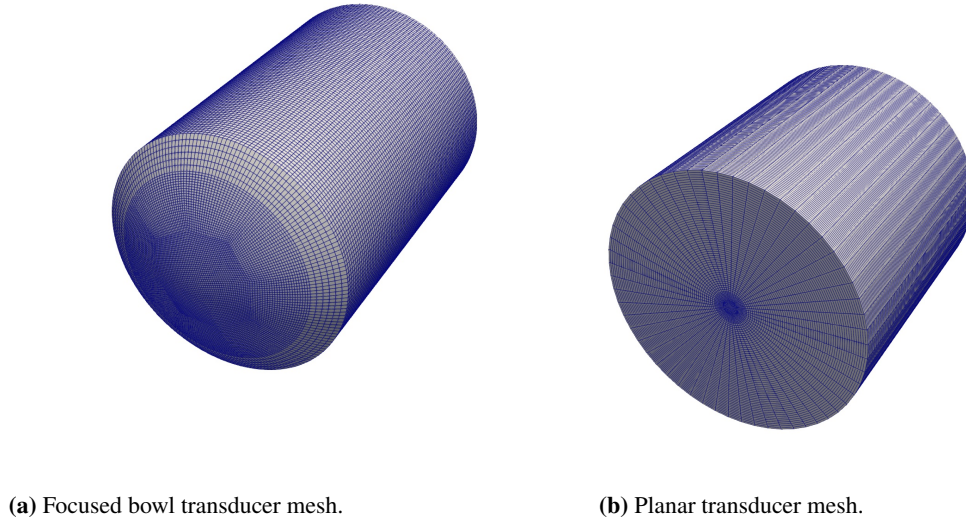


Fig. 6.5 Example meshes used in the simulation.

6.2.5 Post-processing methodology

The FEniCSx-FUS solver produces output in the form of pressure fields. To obtain the pressure amplitude field, the steady-state outputs for a single period are collected at a collection of grid points. To ensure that the wave has achieved steady state, the pressure field is collected after the wave has propagated across the domain plus 6 period cycles such that

$$t_{\text{collect}} > \frac{L}{c_0} + \frac{6}{f_0} \quad (6.6)$$

where L and f_0 are the domain length measured in the direction of the wave propagation and source frequency, respectively. The steady-state output is collected for a single period. To avoid spectral leakage, the time step is set to give an integer number of steps, while satisfying the CFL condition. This is achieved by adjusting the time step size obtained from equation (6.5) such that

$$\Delta t_{\text{adj}} = \frac{T}{N_s} \quad (6.7)$$

where T and N_s denote the period and number of steps per period, respectively. Finally, the fast Fourier transform (FFT) is performed on the pressure field output to obtain the amplitude pressure field. It is important to note here that $\Delta t_{\text{adj}} < \Delta t$.

6.2.6 Comparison metrics

For consistency, the difference metrics used in [16] are considered. The mathematical definitions of the metrics are reproduced here and are shown in table 6.2. The difference metrics compare the overall amplitude pressure field and the focal characteristics. The focal characteristics comparison includes the position and the magnitude of the peak pressure of the focal region. The code to compute the difference metrics is available on GitHub [158].

Table 6.2 Difference metrics used for the comparison between the FEniCSx-FUS solver with the k-Wave solver. The symbols p_f and p_k represent the pressure amplitude field for the FEniCSx-FUS solver and the k-Wave solver sampled at specific grid points, respectively. Sum and maximum values are assumed to be over all values in the comparison domain, starting from the exit plane of the transducers [16]. The function pos max denotes the position of the maximum pressure amplitude.

Metric	Definition
L^2 difference	$\sqrt{\frac{\sum (p_k - p_f)^2}{\sum p_k^2}}$
L^∞ difference	$\frac{\max p_k - p_f }{\max(p_k)}$
Focal (peak) pressure	$\frac{ \max(p_k) - \max(p_f) }{\max(p_k)}$
Focal position	$\ \text{pos max}(p_k) - \text{pos max}(p_f)\ _2$

For benchmarks 1–6, the comparison domain size is 120×70 mm 2D slice through the central z -coordinate with 0.5 mm grid sampling, which corresponds to 241×141 grid points. For benchmark 7, the comparison domain size is $120 \times 70 \times 70$ mm, which corresponds to $241 \times 141 \times 141$ grid points.

6.2.7 Mesh convergence study

A mesh convergence study was performed to validate the correctness of the solver. To this end, the scattering by a homogeneous sphere in water due to a circular planar piston source is considered. The circular planar piston source has a radius of 1 m, a frequency of 2 kHz, with speed of 1 ms^{-1} . The

sphere is made up of a cortical bone material with a radius of 1 m. This particular problem was chosen due to the availability of the OptimUS [69, 168, 171, 170] solver that serves as a reference to the output of the FEniCSx-FUS solver. The OptimUS solver's element size is set to 2 cm, corresponding to approximately 37 elements per wavelength. This resolution is considered sufficient for an accurate representation, as a study has shown that the solver is accurate with more than 5 elements per wavelength [16]. The total number of nodes is about 49×10^3 . The material properties used are taken from table 6.1. The problem layout is shown in figure 6.6.

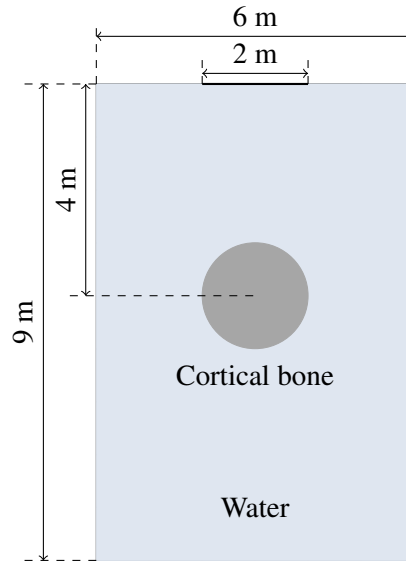


Fig. 6.6 Source locations and cross-sectional view for the scattering by a sphere problem. The x -axis is set as the axial position, the y and z -axis are set as the lateral position. The axial length is 9 m and the lateral length are 6 m. The radius of the circular planar piston and the spherical scatterer is 1 m.

Figure 6.7 shows the convergence of the L^2 and L^∞ relative differences as the number of elements per wavelength increases. The output from the FEniCSx-FUS solver was compared with the OptimUS solver. Both relative differences were observed to decrease with increasing number of elements per wavelength, reaching a value of approximately 0.1 between 2 to 3 elements per wavelength. Beyond 3 elements per wavelength, the relative differences remain consistent. This validates the accuracy of the FEniCSx-FUS solver and demonstrates that a resolution of 2 to 3 elements per wavelength is sufficient to achieve a reliable solution. Figure 6.8 presents the convergence of pressure amplitudes with increasing mesh resolution. The FEniCSx-FUS outputs were observed to visually converge to the OptimUS outputs as the number of elements increased.

6.2.8 Benchmarks 1 and 2

Benchmark 1 consists of a homogeneous domain (water) with no absorption, i.e. $\delta = 0$. Benchmark 2 adds attenuation to benchmark 1 with the material properties taken from table 6.1, i.e. 1 dB cm^{-1} at 500 kHz. The sound attenuation for water in benchmark 2 is deliberately set to be large to verify that

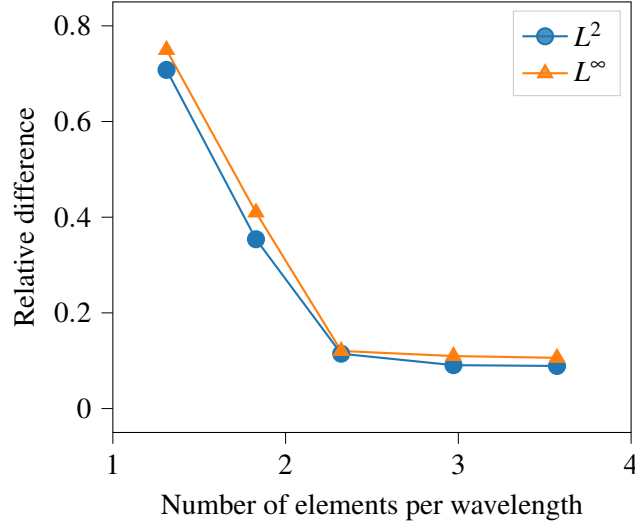


Fig. 6.7 Convergence of the L^2 and L^∞ relative differences for increasing number of elements per wavelength. The L^2 and L^∞ relative differences are based on table 6.2.

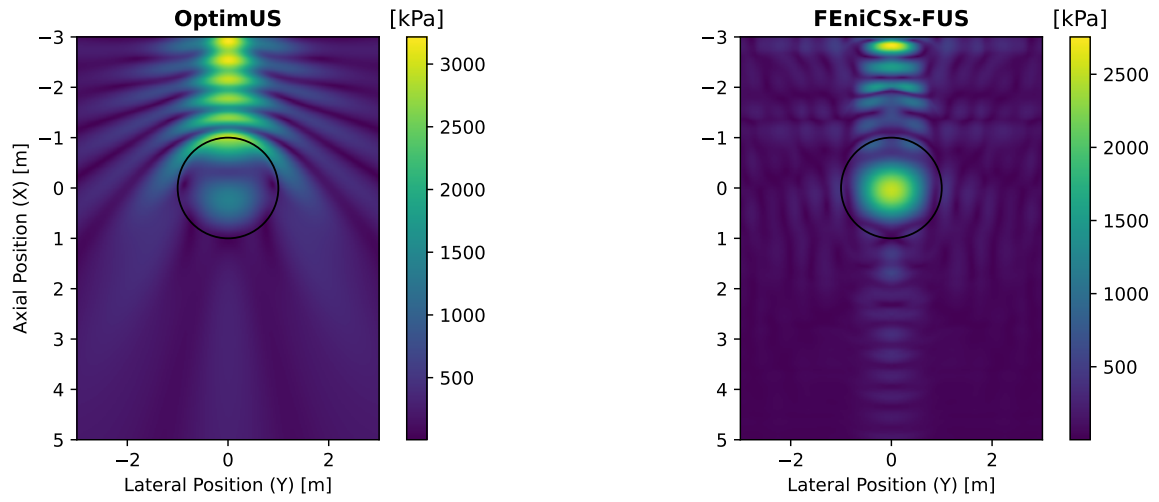
the solver implements the attenuation correctly. The layouts for benchmarks 1 and 2 are shown in figure 6.9.

Figures 6.10 and 6.11 present the pressure amplitudes and profiles for benchmark 1 of the FEniCSx-FUS solver as compared to the k-Wave solver, respectively. In both source configurations, good agreements are observed between the two solvers. In fact, the L^∞ difference between the FEniCSx-FUS and k-Wave for benchmark 1 is 2.7%, with the largest differences are close to the transducer surface as this is where the pressure is varying rapidly. For the focused bowl transducer, the focal peak position is the same while the focal peak pressure difference is less than 1%. Accordingly, this verify that the transducer configuration has been correctly implemented.

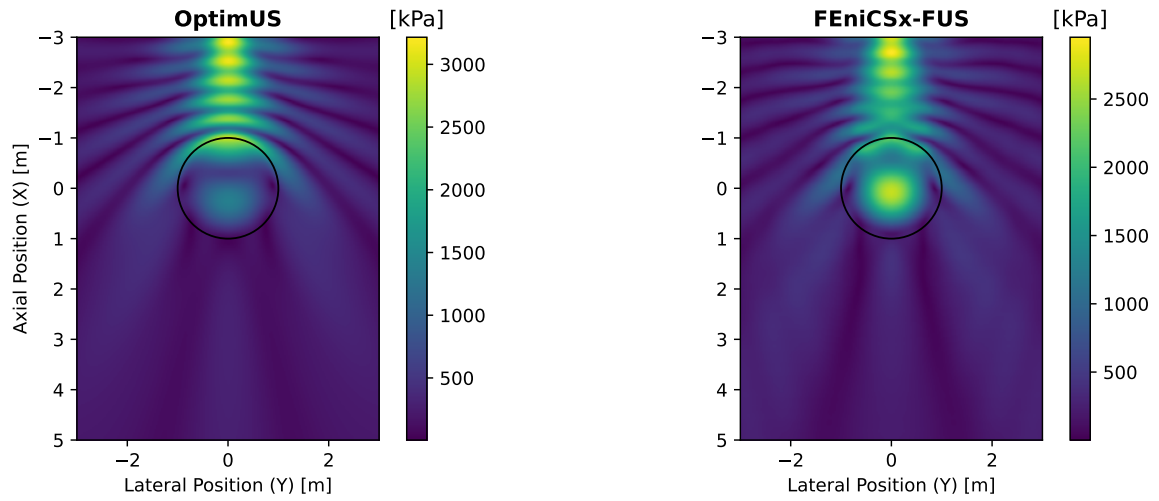
For the focused bowl source, the FEniCSx-FUS simulation was performed on the Intel Icelake CPU using 260 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 62×10^6 . The simulation took around 9 minutes (8700 time steps), which includes saving a 2D slice of a single period to file. For the planar source, the FEniCSx-FUS simulation was performed on the Intel Icelake CPU using 62 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 31×10^6 . The simulation took around 4 minutes (2300 time steps), which includes saving a 2D slice of a single period to file.

Figures 6.12 and 6.13 show the pressure amplitudes and profiles for benchmark 2. Good agreements are observed between the FEniCSx-FUS solver with k-Wave. In fact, the difference in the focal peak pressure between FEniCSx-FUS and k-Wave is less than 1% for the focused bowl transducer. It can also be seen that the maximum amplitude for benchmark 2 is significantly lower than in benchmark 1.

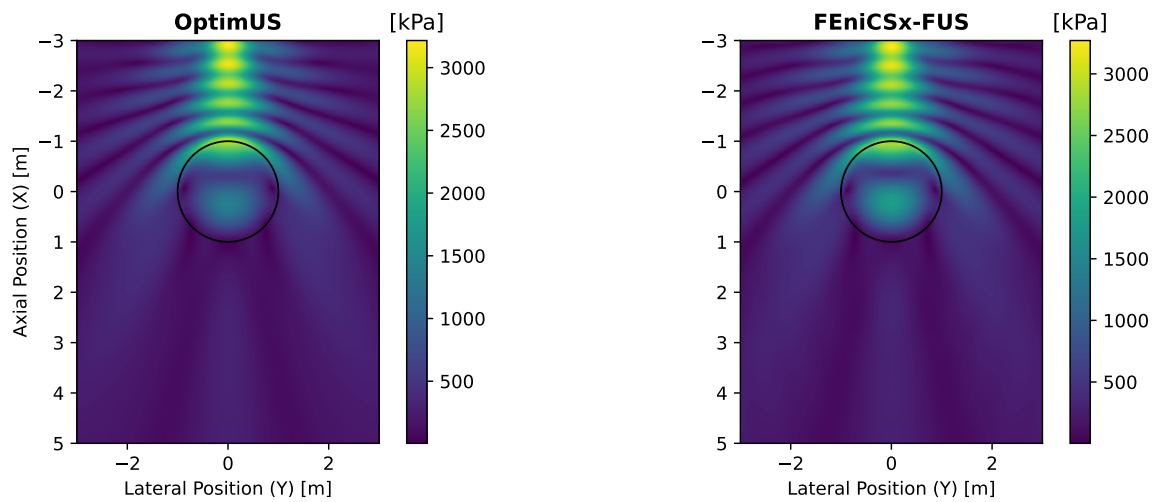
For the focused bowl source, the FEniCSx-FUS simulation was performed on the Intel Icelake CPU using 288 processes with single precision floating-point number. The degrees-of-freedom of



(a) 2046 hexahedral elements.



(b) 6738 hexahedral elements.

(c) 2.1×10^4 hexahedral elements.

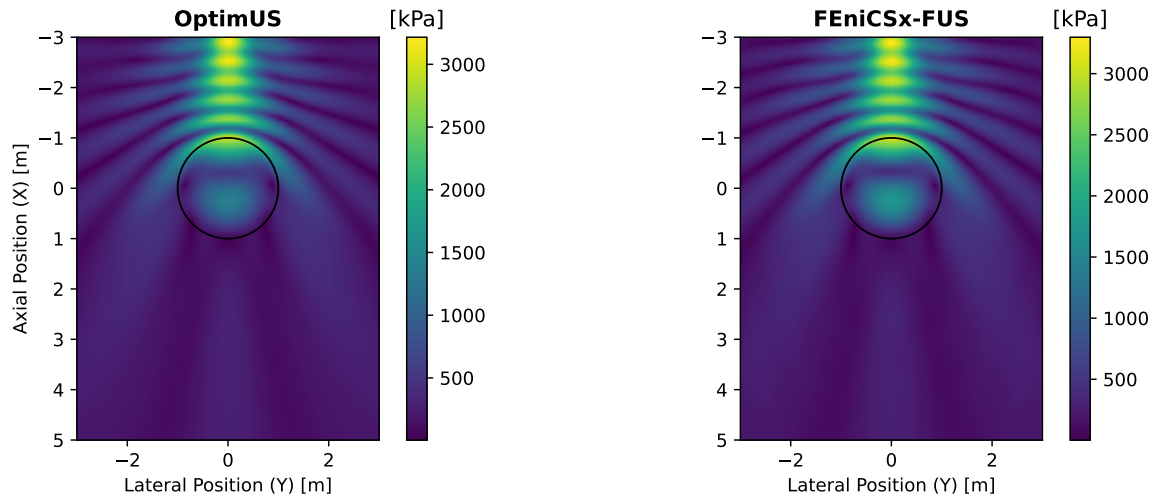
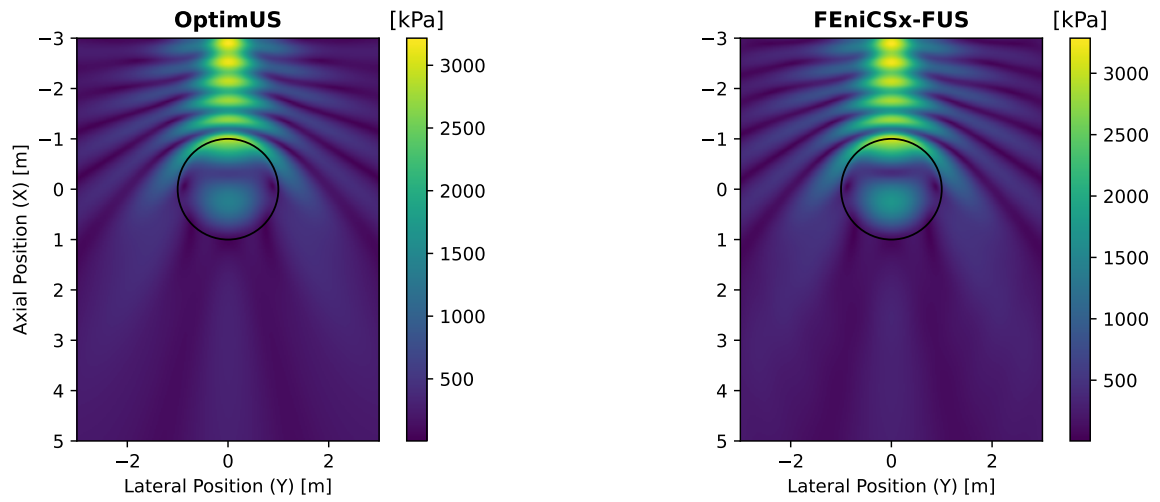
(d) 9.4×10^4 hexahedral elements.(e) 3.7×10^5 hexahedral elements.

Fig. 6.8 A comparison of the pressure amplitude between OptimUS (left) and FEniCSx-FUS (right) solver. The comparison domain size is 6×8 m 2D slice with 250×250 grid points. Observed that above 20000 elements, the output of both solver is visually similar.

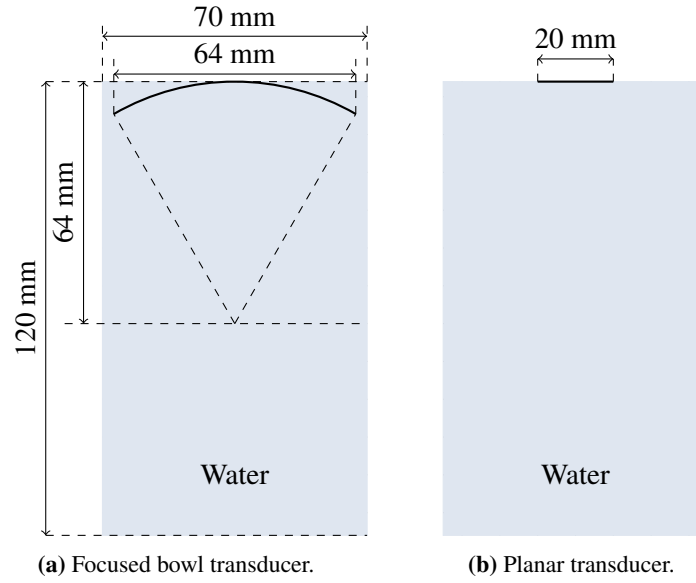
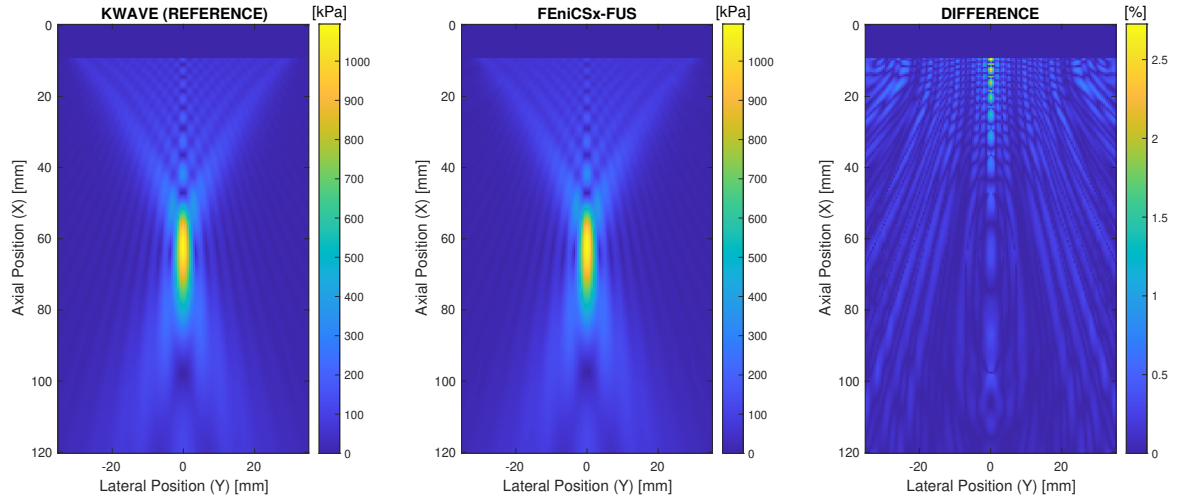
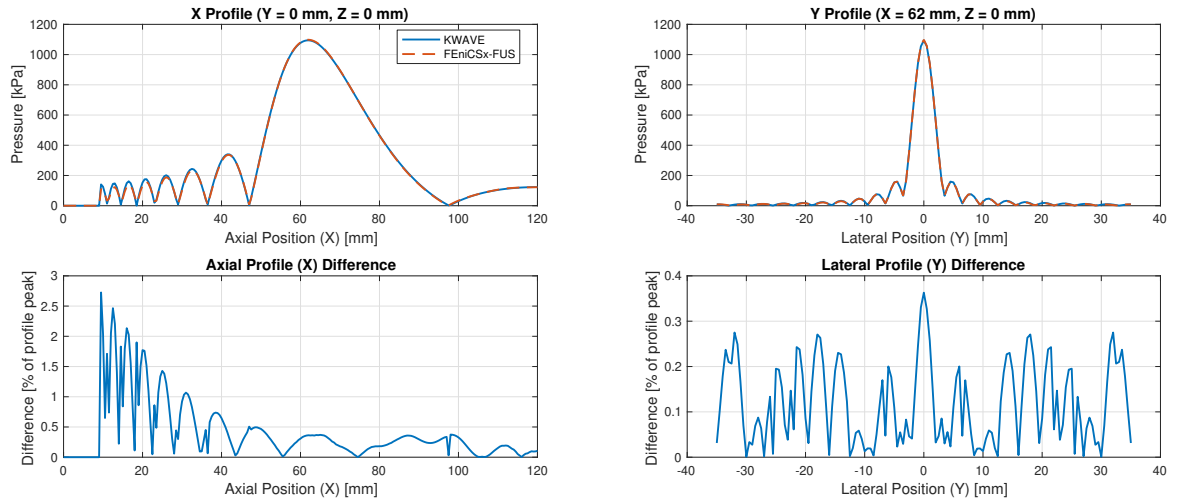


Fig. 6.9 Source locations and cross-sectional view for benchmark 1 and 2. The x -axis is set as the axial position, the y and z -axis are set as the lateral position. The axial length is 120 mm and the lateral length are 70 mm. Reproduced from [16].

the simulation is approximately 62×10^6 . The simulation took around 13 minutes (9600 time steps), which includes saving a 2D slice of a single period to file. For the planar source, the FEniCSx-FUS simulation was performed on the Intel Icelake CPU using 62 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 31×10^6 . The simulation took around 6 minutes (2300 time steps), which includes saving a 2D slice of a single period to file.

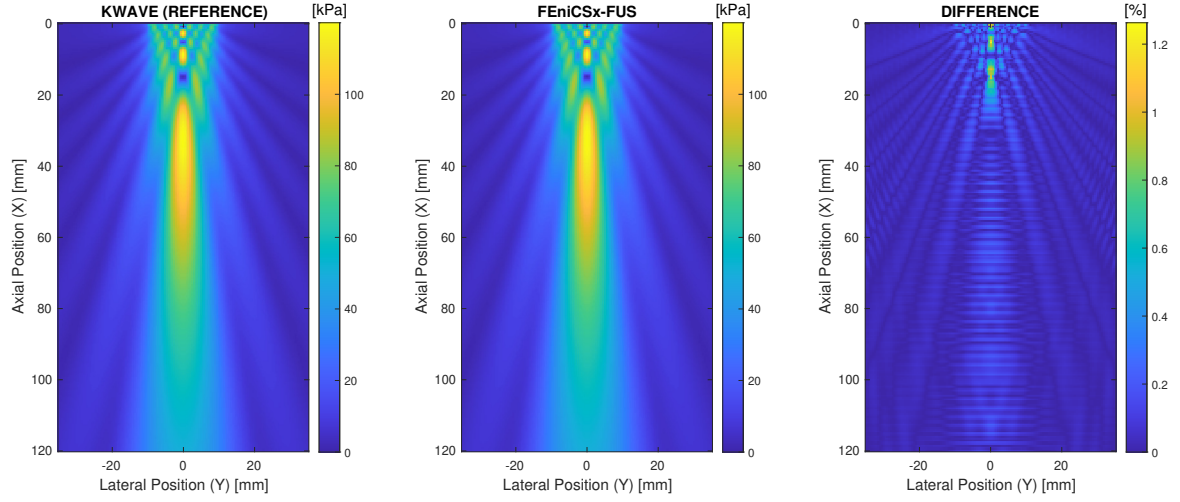


(a) The pressure amplitudes for benchmark 1 with focused bowl source.

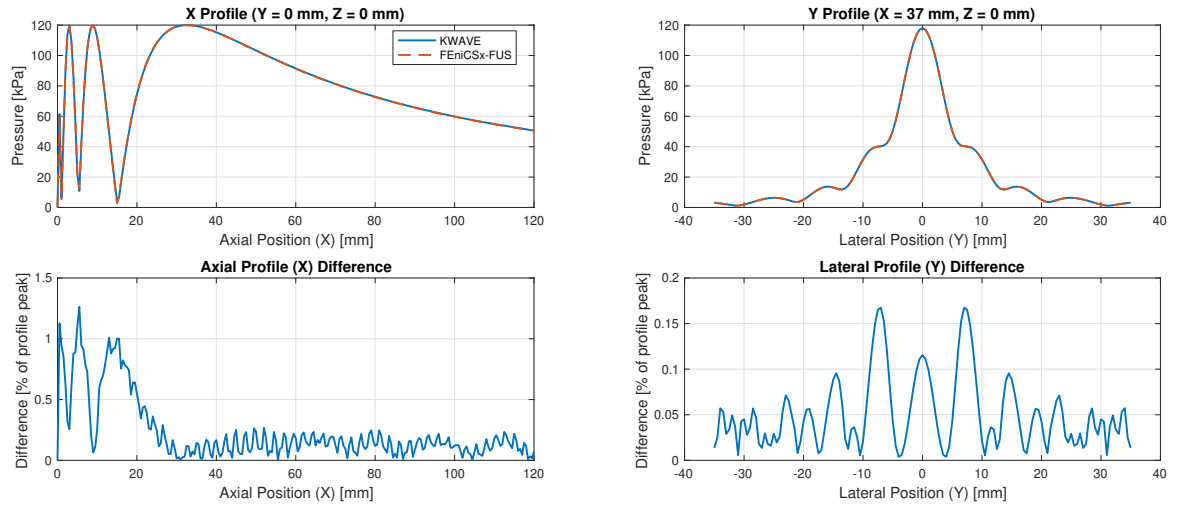


(b) The profiles for benchmark 1 with focused bowl source.

Fig. 6.10 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 1 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.

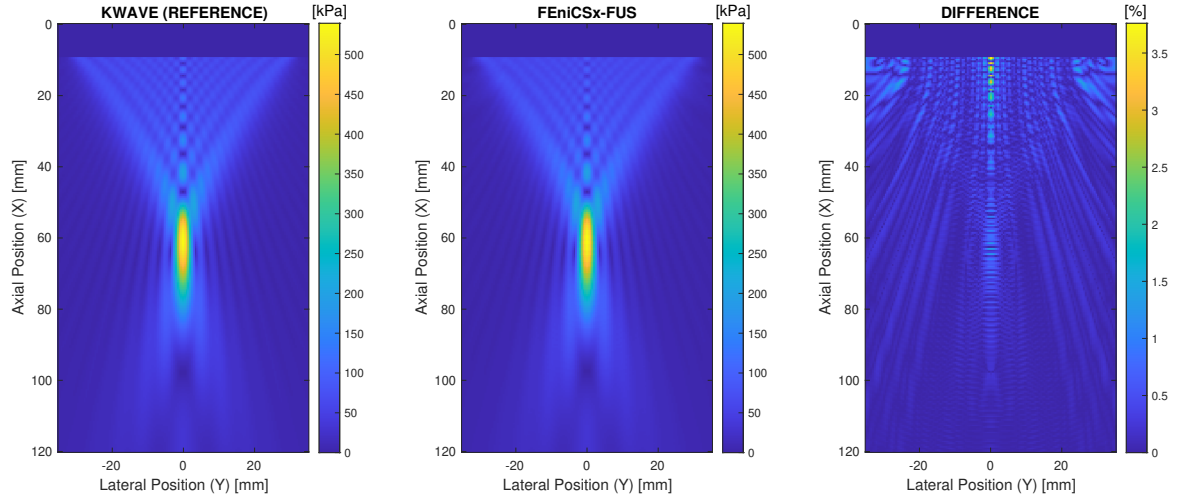


(a) The pressure amplitudes for benchmark 1 with planar source.

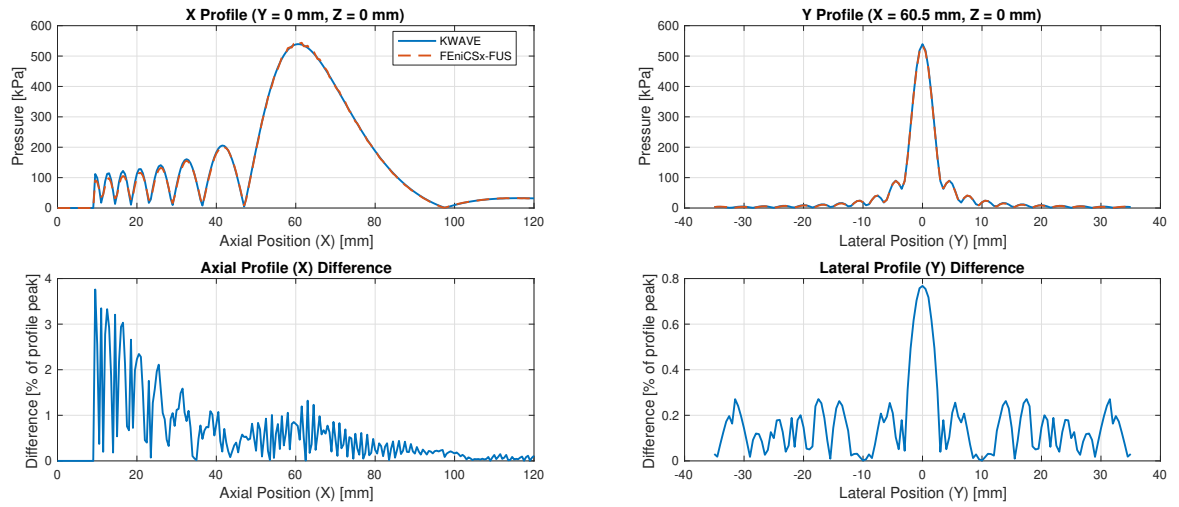


(b) The profiles for benchmark 1 with planar source.

Fig. 6.11 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 1 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.

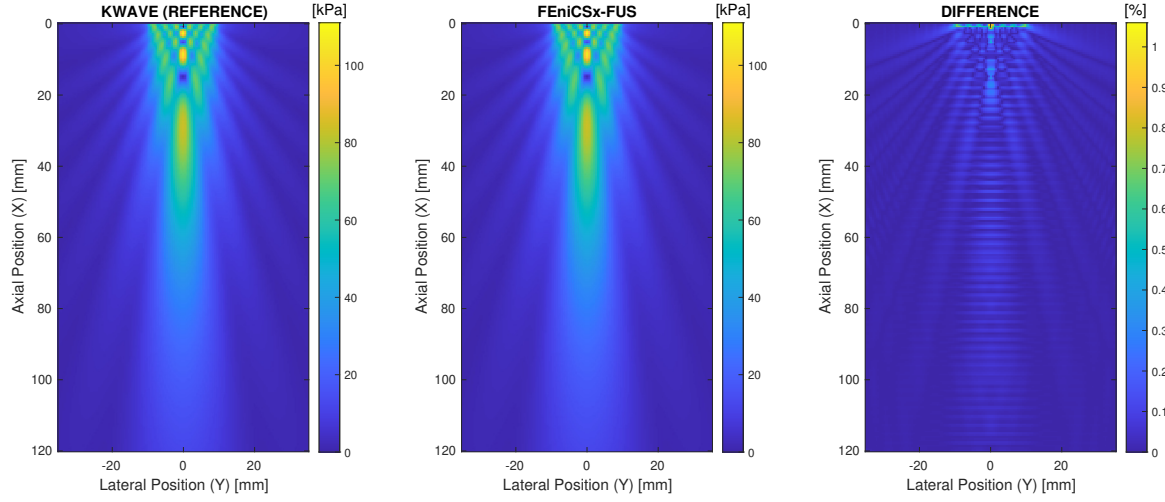


(a) The pressure amplitudes for benchmark 2 with focused bowl source.

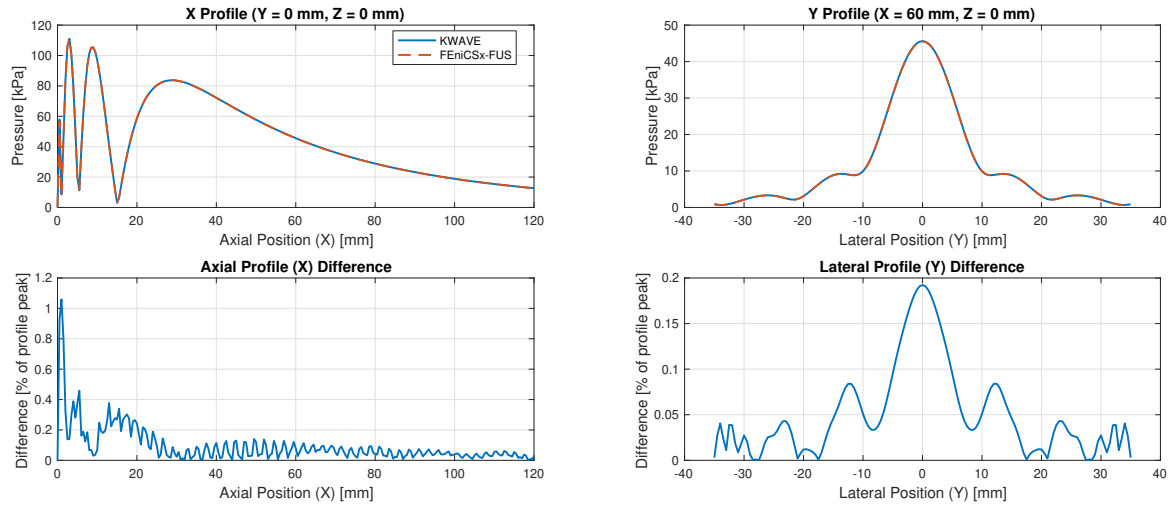


(b) The profiles for benchmark 2 with focused bowl source.

Fig. 6.12 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 2 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.



(a) The pressure amplitudes for benchmark 2 with planar source.



(b) The profiles for benchmark 2 with planar source.

Fig. 6.13 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 2 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.

6.2.9 Benchmark 3

Benchmark 3 increases complexity by including a slab of cortical bone, as shown in figure 6.14. The slab of cortical bone of 6.5 mm thickness is position at 30 mm from the transducer centre. The material properties of the cortical bone are given in table 6.1.

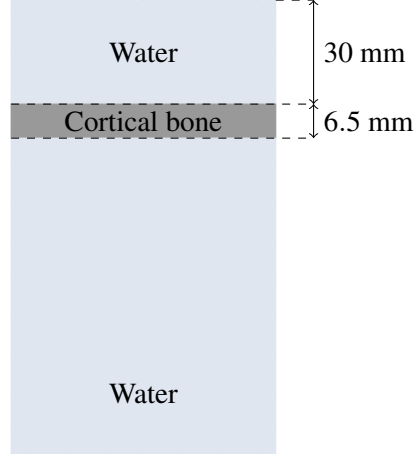
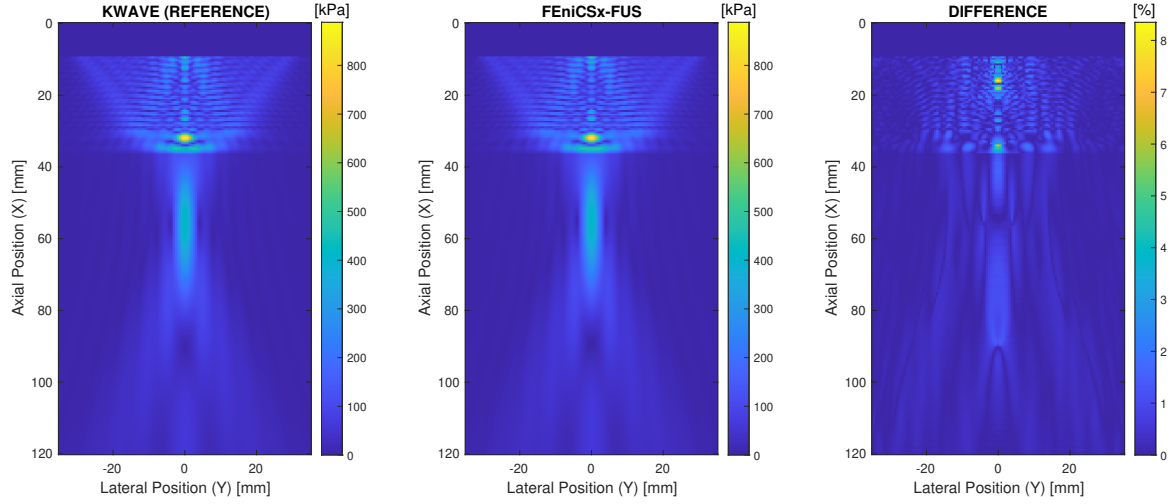


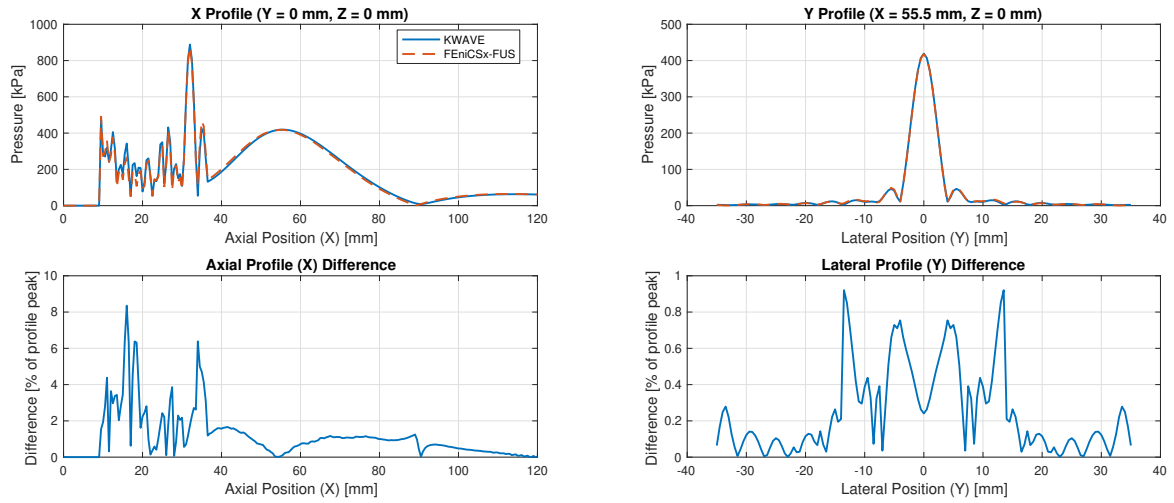
Fig. 6.14 Layout for benchmark 3. Reproduced from [16]. The material properties used are given in table 6.1.

Figures 6.15 and 6.16 show the pressure amplitudes and profiles for benchmark 3. For both transducer configurations, good agreement is observed between the FEniCSx-FUS solver with the k-Wave solver, with the maximum difference of less than 10%, which is along the near-field close to the source. At the focus and away from the source, close agreement is obtained between the two solvers. Unlike benchmark 1 and 2, the presence of the slab of cortical bone creates reflection at the interface between the water and cortical bone, thus, changes the maximum pressure position. This emphasises the importance of running a simulation to prevent unwanted heating at wrong locations. The close agreement with k-Wave pressure profile validates that the FEniCSx-FUS solver can handle domain heterogeneity.

For the focused bowl source, the FEniCSx-FUS simulation was performed on the Intel Icelake CPU using 320 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 61×10^6 . The simulation took around 37 minutes (32000 time steps), which includes saving a 2D slice of a single period to file. For the planar source, the FEniCSx-FUS simulation was performed on the Intel Icelake CPU using 189 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 32×10^6 . The simulation took around 12 minutes (12300 time steps), which includes saving a 2D slice of a single period to file.

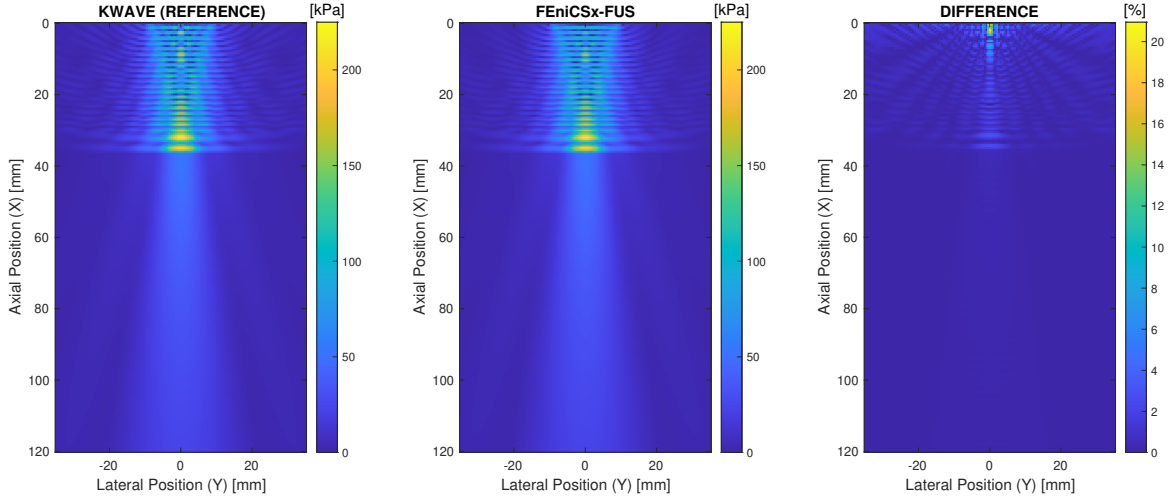


(a) The pressure amplitudes for benchmark 3 with focused bowl source.

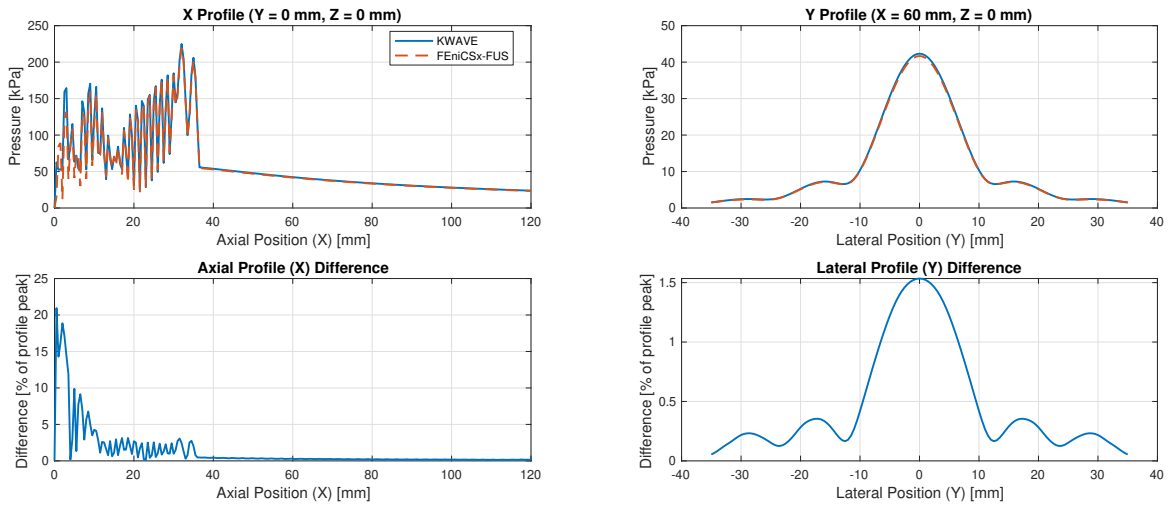


(b) The profiles for benchmark 3 with focused bowl source.

Fig. 6.15 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 3 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.



(a) The pressure amplitudes for benchmark 3 with planar source.



(b) The profiles for benchmark 3 with planar source.

Fig. 6.16 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 3 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.

6.2.10 Benchmark 4

Benchmark 4 extends benchmark 3 by adding more material heterogeneity, as shown in figure 6.17. The domain consists of water, a skin layer, three-layered skull (made up of a layer of trabecular bone and two layers of cortical bone), and brain. The material properties are given in table 6.1.

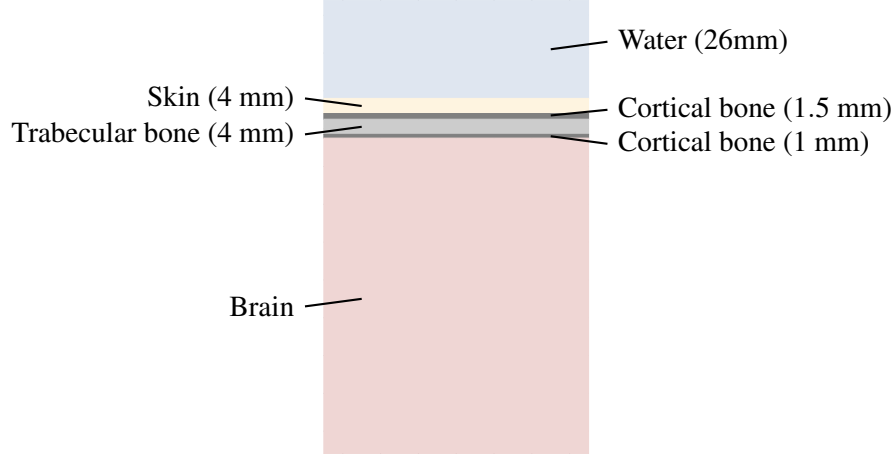
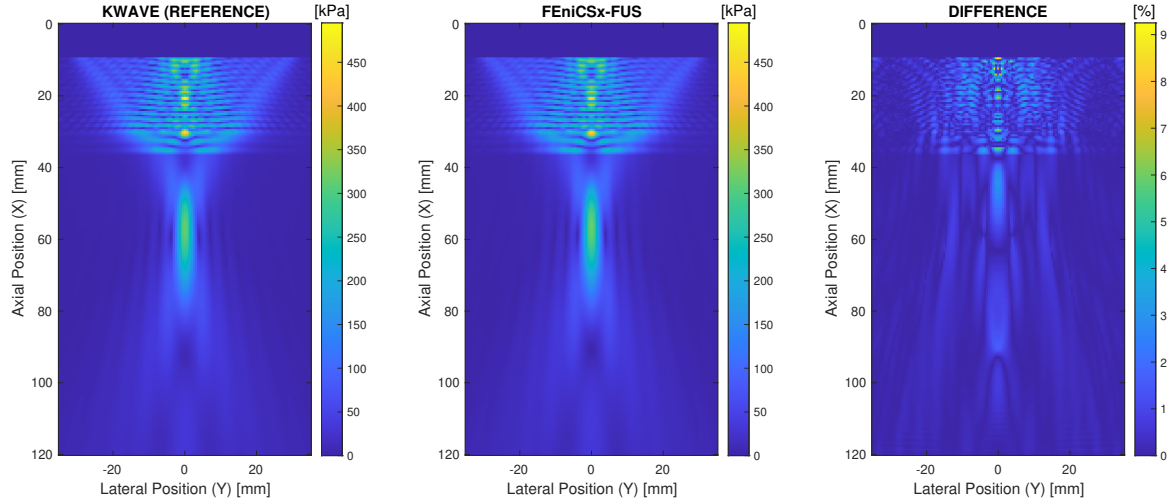


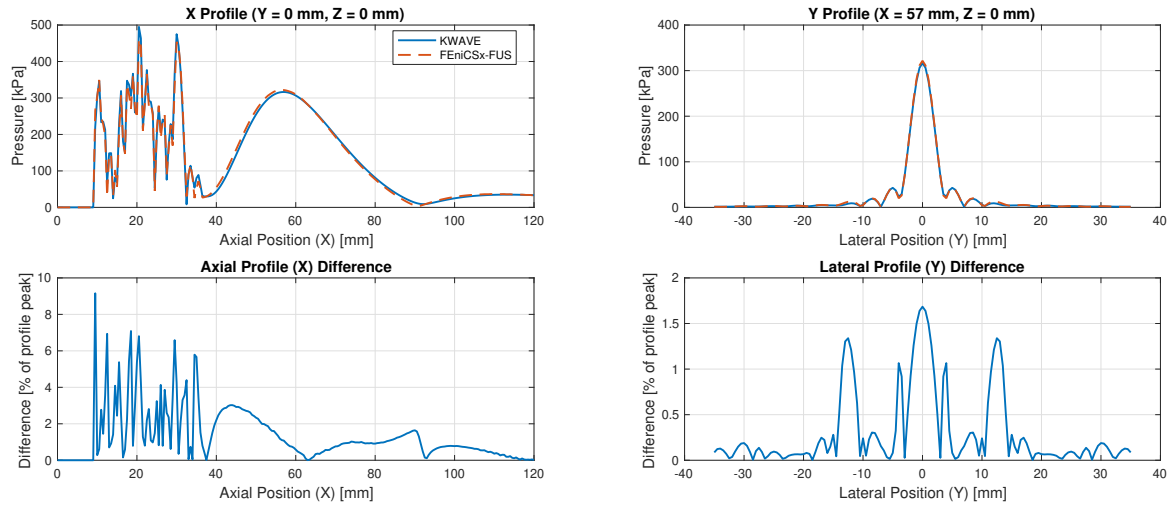
Fig. 6.17 Layout for benchmark 4. Reproduced from [16]. The material properties used are given in table 6.1.

Figures 6.18 and 6.19 show the computed pressure amplitudes and profiles for benchmark 4. Good agreements are observed between the FEniCSx-FUS and the k-Wave solver. This shows that the FEniCSx-FUS solver can handle multiple material heterogeneities. Despite some differences near the source, the difference in the pressure field at the focus location and away from the source are small (less than 10%). Additionally, for the focused bowl source, the location of the maximum amplitude is exact and the difference in the focal pressure is less than 2%.

For the focused bowl source, the FEniCSx-FUS simulation was performed on the Intel Icelake CPU using 315 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 60×10^6 . The simulation took around 38 minutes (32000 time steps), which includes saving a 2D slice of a single period to file. For the planar source, the FEniCSx-FUS simulation was performed on the Intel Icelake CPU using 189 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 32×10^6 . The simulation took around 15 minutes (14000 time steps), which includes saving a 2D slice of a single period to file.

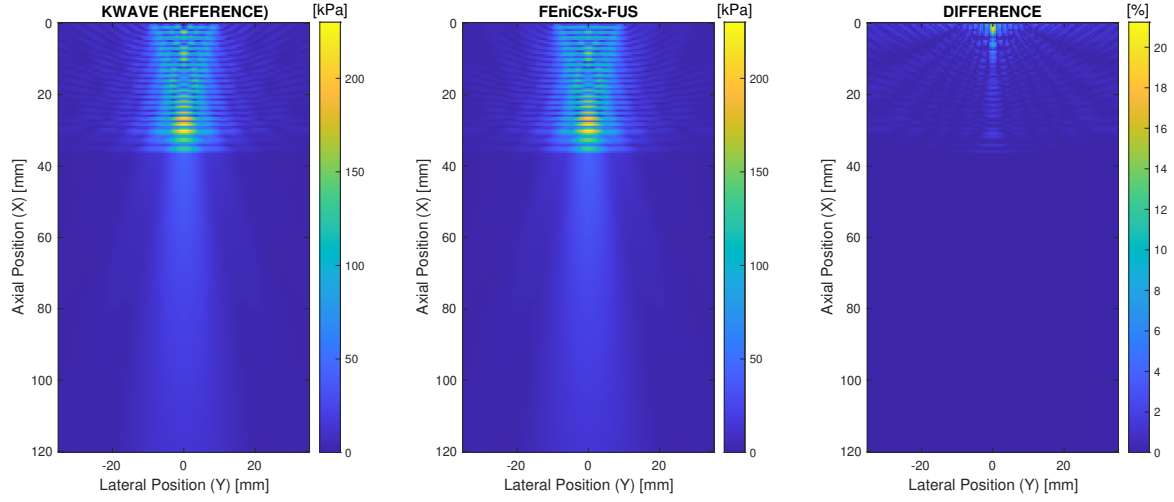


(a) The pressure amplitudes for benchmark 4 with focused bowl source.

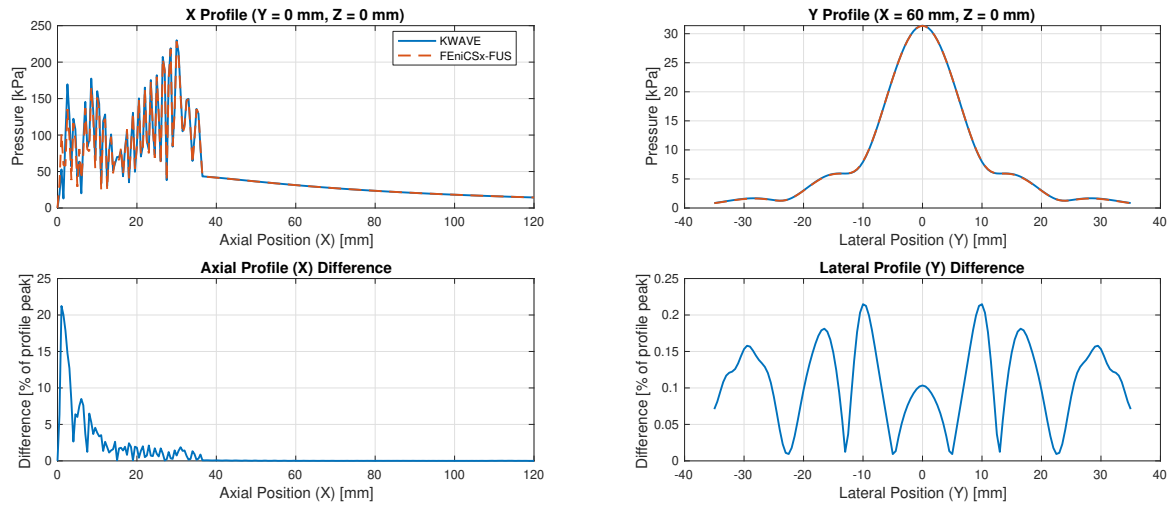


(b) The profiles for benchmark 4 with focused bowl source.

Fig. 6.18 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 4 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.



(a) The pressure amplitudes for benchmark 4 with planar source.



(b) The profiles for benchmark 4 with planar source.

Fig. 6.19 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 4 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.

6.2.11 Benchmark 5

Benchmark 5 is geometrically more complex by introducing a spherical curve cortical bone layer as shown in figure 6.20. The cortical bone layer has a thickness of 6.5 mm, an outer radius of curvature of 75 mm and an inner radius of curvature of 68.5 mm. The material properties are taken from table 6.1.

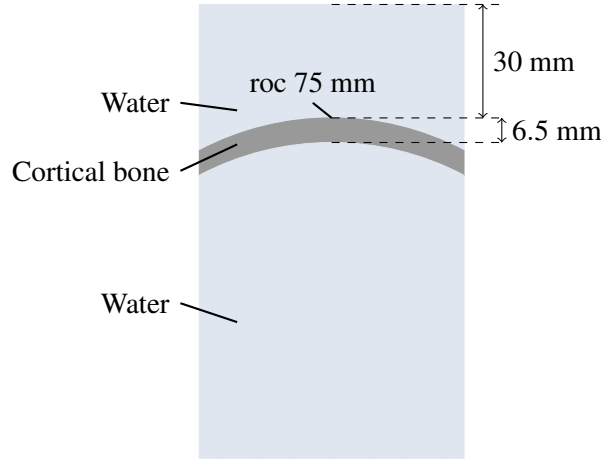
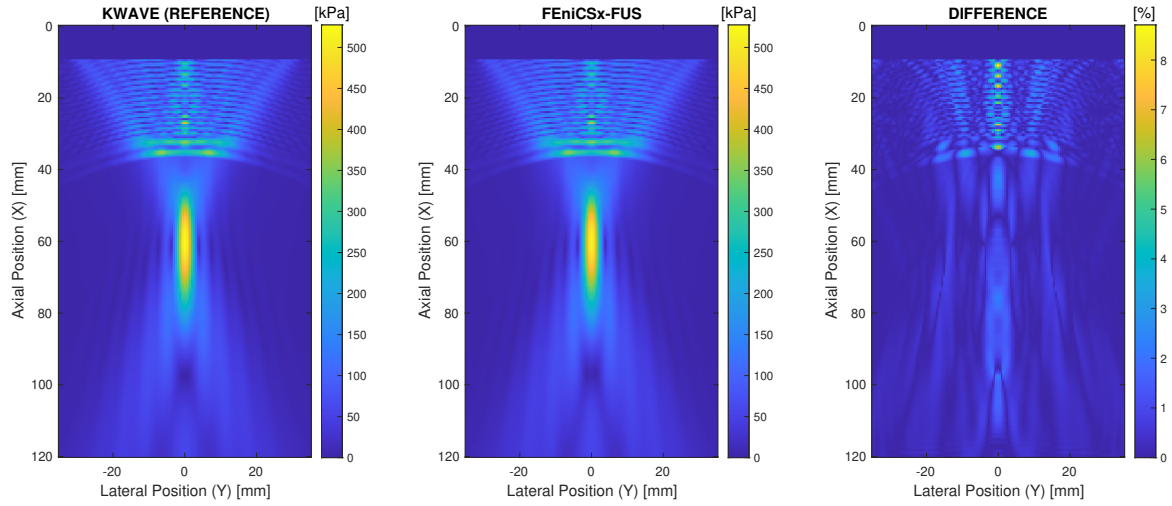


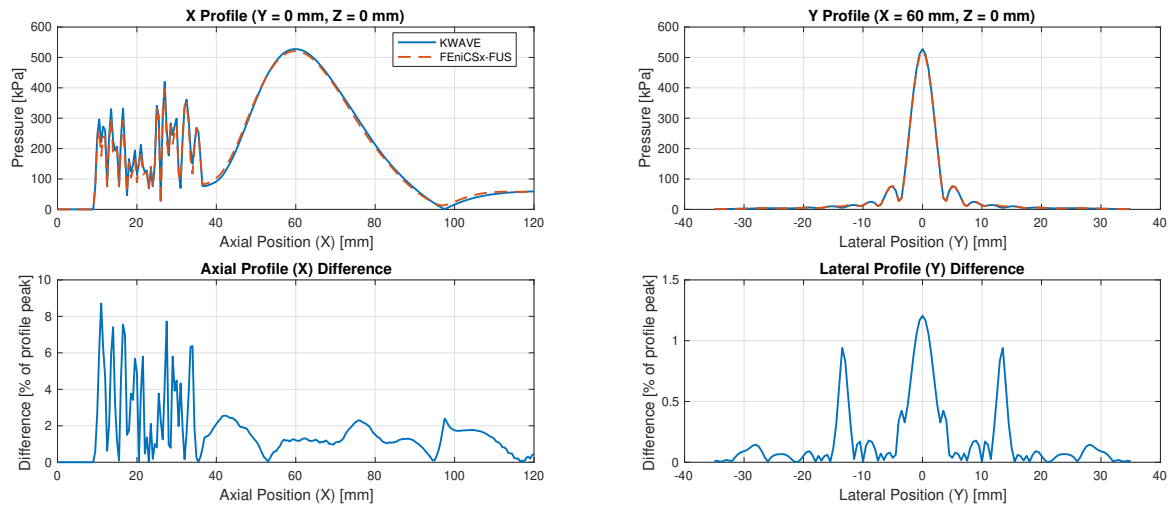
Fig. 6.20 Layout for benchmark 5. Reproduced from [16]. The material properties used are given in table 6.1.

Figures 6.21 and 6.22 show the pressure amplitude and profiles for benchmark 5. Like benchmark 3, close agreement is observed between the FEniCSx-FUS solver as compared to the k-Wave solver. With regard to the focused bowl transducer, the largest difference is close to the source where the maximum difference is around 9%. However, away from the source and at the focus region, almost exact agreement is observed whereby the difference is less than 4%. Additionally, the focal position is exact and the difference in the peak pressure is less than 2%. This shows that the FEniCSx-FUS solver can handles different geometric shape.

For the focused bowl source, the FEniCSx-FUS simulation was performed on the Intel Skylake CPU using 330 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 72×10^6 . The simulation took around 1 hour 40 minutes (70000 time steps), which includes saving a 2D slice of a single period to file. For the planar source, the FEniCSx-FUS simulation was performed on the Intel Skylake CPU using 186 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 27×10^6 . The simulation took around 14 minutes (13800 time steps), which includes saving a 2D slice of a single period to file.

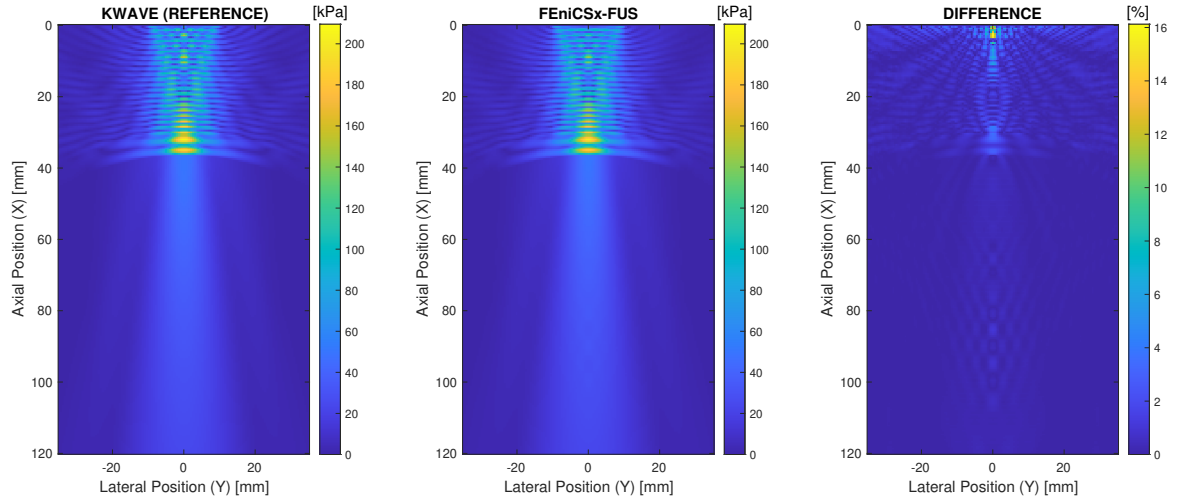


(a) The pressure amplitudes for benchmark 5 with focused bowl source.

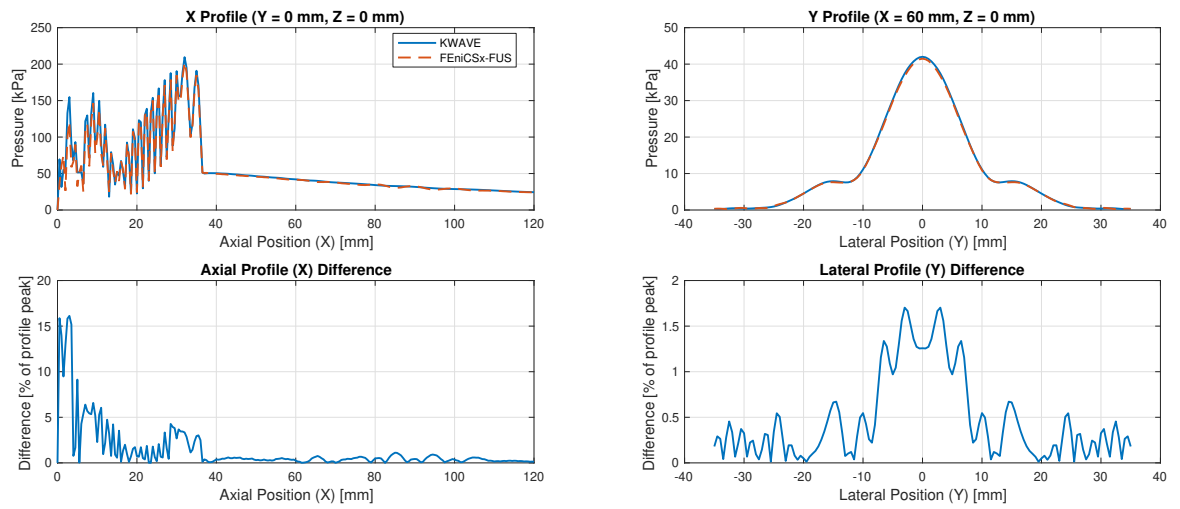


(b) The profiles for benchmark 5 with focused bowl source.

Fig. 6.21 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 5 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.



(a) The pressure amplitudes for benchmark 5 with planar source.



(b) The profiles for benchmark 5 with planar source.

Fig. 6.22 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 5 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.

6.2.12 Benchmark 6

Similar to benchmark 5, benchmark 6 extends benchmark 4 by changing the shape of the material layers. In particular, the skin layer and the multilayer bones are set to be spherical shape with the same geometry as in benchmark 5, the layout is shown in figure 6.23. The material properties are given in table 6.1.

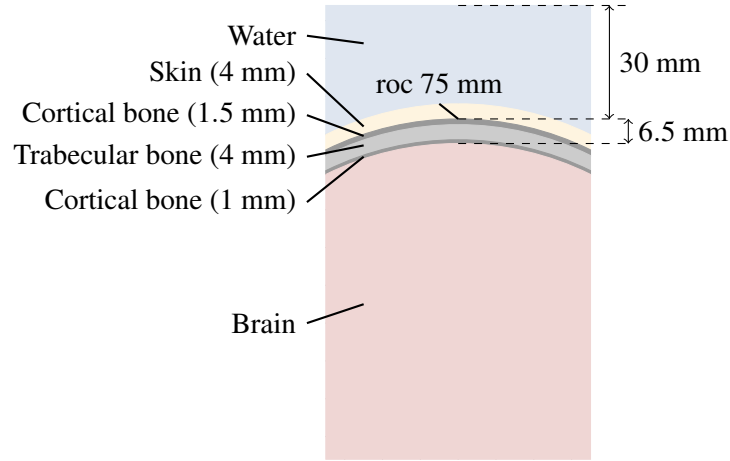
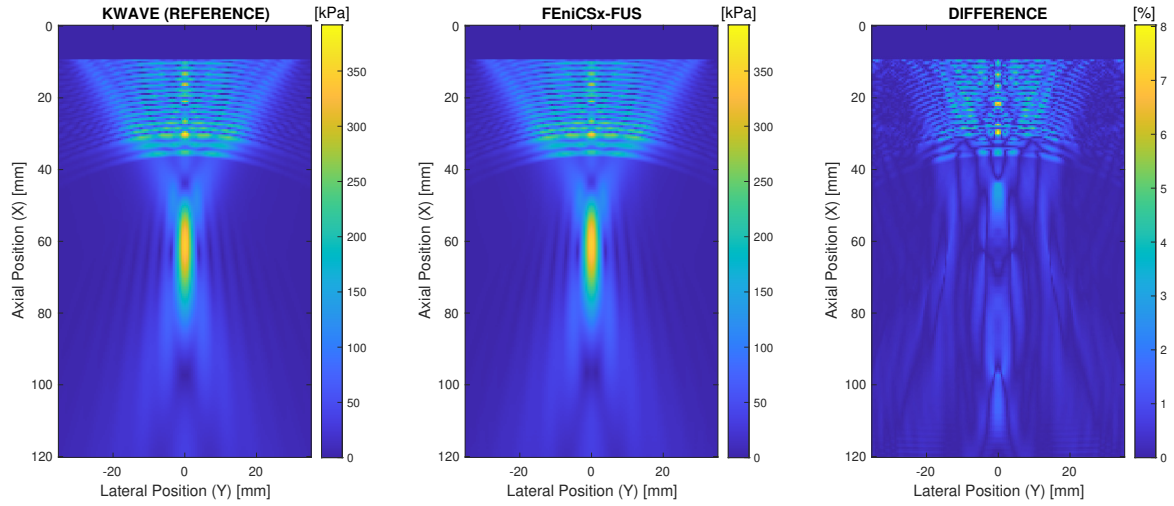


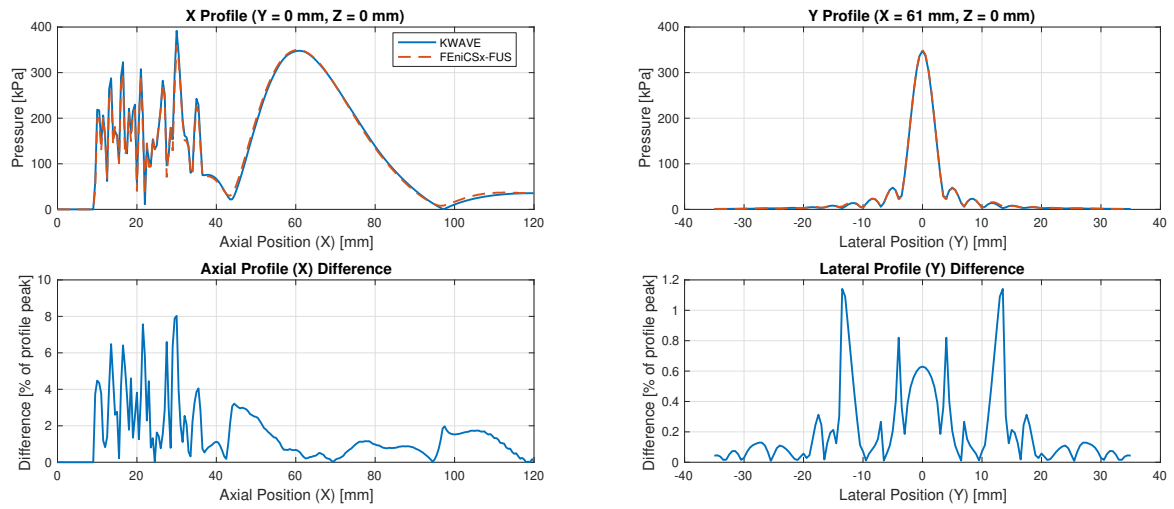
Fig. 6.23 Layout for benchmark 6. Reproduced from [16]. The material properties used are given in table 6.1.

Figures 6.24 and 6.25 show the pressure amplitude and profiles for benchmark 6. For both transducer configuration, close agreements are observed between the FEniCSx-FUS solver and the k-Wave solver, with the largest difference seen in the piston transducer close to the source surface. Nonetheless, away from the transducer surface and at the focal region, the difference between the two solvers is less than 10%. For the focused bowl transducer, the peak pressure difference is less than 1% and the focal position differs by 0.5 mm. This validates that the FEniCSx-FUS solver can handle multilayer materials with non-trivial shapes.

For the focused bowl source, the FEniCSx-FUS simulation was performed on the Intel Skylake CPU using 256 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 71×10^6 . The simulation took around 1 hour (34000 time steps), which includes saving a 2D slice of a single period to file. For the planar source, the FEniCSx-FUS simulation was performed on the Intel Skylake CPU using 186 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 27×10^6 . The simulation took around 14 minutes (14400 time steps), which includes saving a 2D slice of a single period to file.

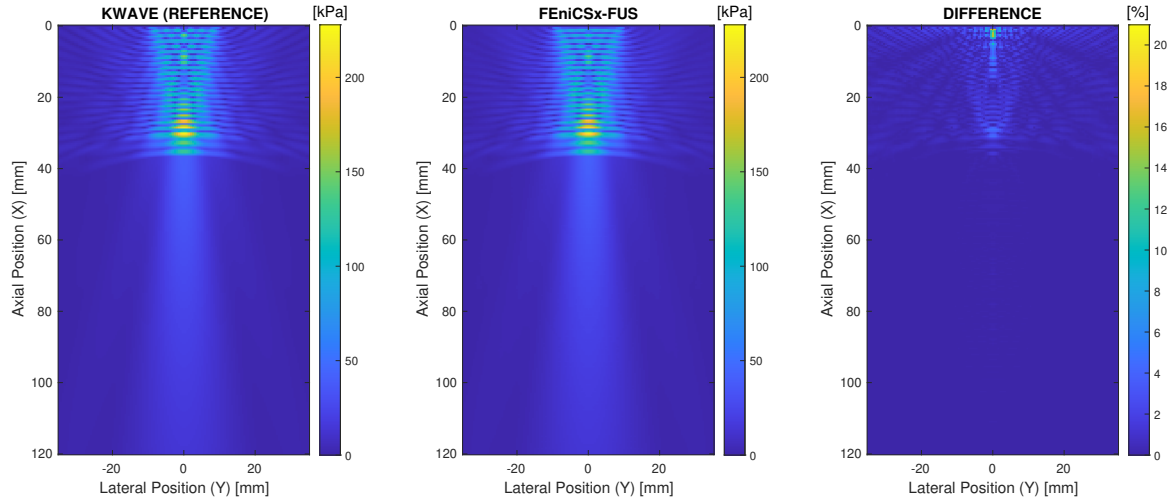


(a) The pressure amplitudes for benchmark 6 with focused bowl source.

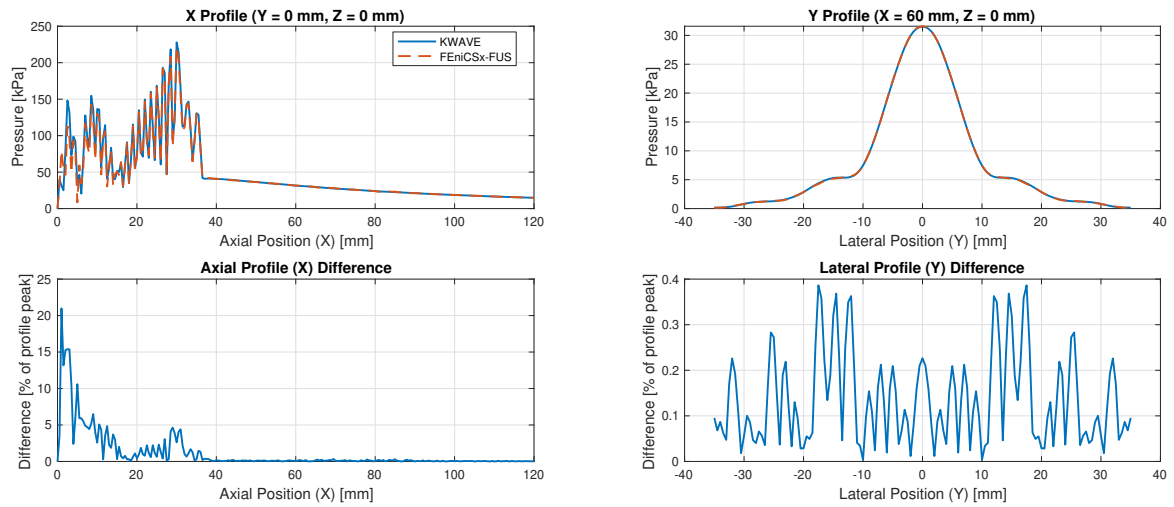


(b) The profiles for benchmark 6 with focused bowl source.

Fig. 6.24 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 6 with focused bowl source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.



(a) The pressure amplitudes for benchmark 6 with planar source.



(b) The profiles for benchmark 6 with planar source.

Fig. 6.25 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 6 with planar source. The comparison domain size is 120×70 mm 2D slice with 0.5 mm grid sampling, which corresponds to 241×141 grid points.

6.2.13 Benchmark 7

In contrast to the axisymmetric geometry of the layers considered in the previous benchmarks, benchmark 7 uses a homogeneous skull mesh obtained from MNI152_T1_1 mm magnetic resonance imaging template of the brain [56, 57]. The benchmark considers a partition of the skull as opposed to the full skull since the main aim is to test whether the FEniCSx-FUS solver is capable of handling a realistic simulation setting. A slice of the skull mesh can be viewed in [16]. The skull properties are defined to be the cortical bone as given in table 6.1.

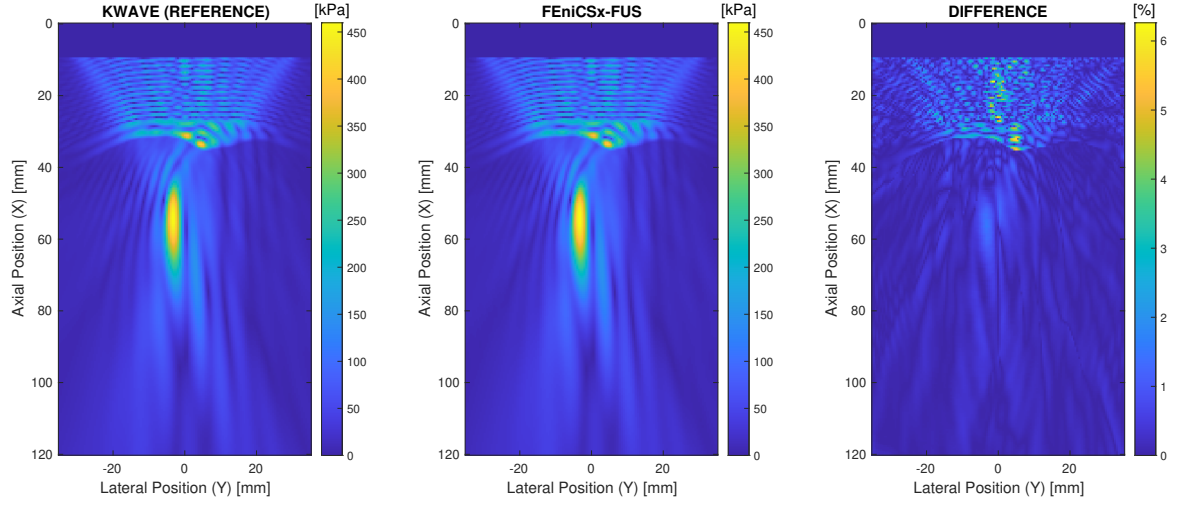
Figures 6.26 and 6.27 show the pressure amplitude and profiles for benchmark 7. In contrast to benchmark 1–6, benchmark 7 is not axisymmetric and as such is more geometrically complex. However, good agreements are observed between the FEniCSx-FUS solver and the k-Wave solver for both source configurations, especially at regions away from the source surface. For the focused bowl transducer, the maximum difference is less than 10%. In addition, the focal peak pressure difference is less than 2% and the focal position between the two solver is the same, which verifies that FEniCSx-FUS solver is suited for realistic simulations.

For the focused bowl source, the FEniCSx-FUS simulation was performed on the Intel Skylake CPU using 416 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 85×10^6 . The simulation took around 6 hours (26800 time steps), which includes saving a 3D volume slice of a single period to file. For the planar source, the FEniCSx-FUS simulation was performed on the Intel Skylake CPU using 448 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 118×10^6 . The simulation took around 5 hours (15400 time steps), which includes saving a 3D volume slice of a single period to file.

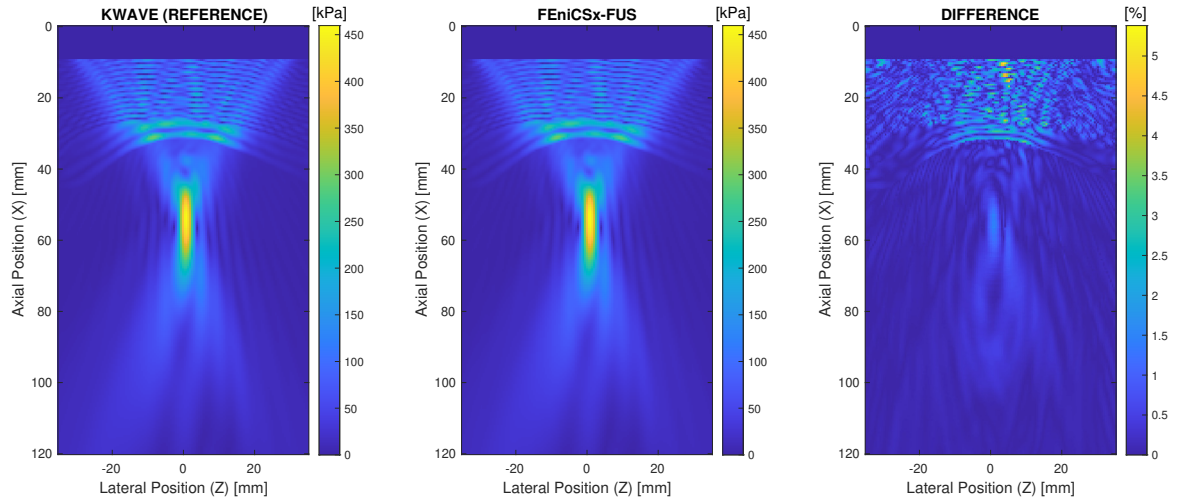
6.2.14 Summary of transcranial focused ultrasound simulations

The benchmarks presented verify that the FEniCSx-FUS solver is correctly implemented and is capable of handling various simulation scenarios. This includes scenarios such as different transducer configurations, heterogeneous materials, and geometrically complex scatterers. Even though good agreements were obtained for all benchmark problems as compared with the k-Wave solver, there was still some small discrepancies.

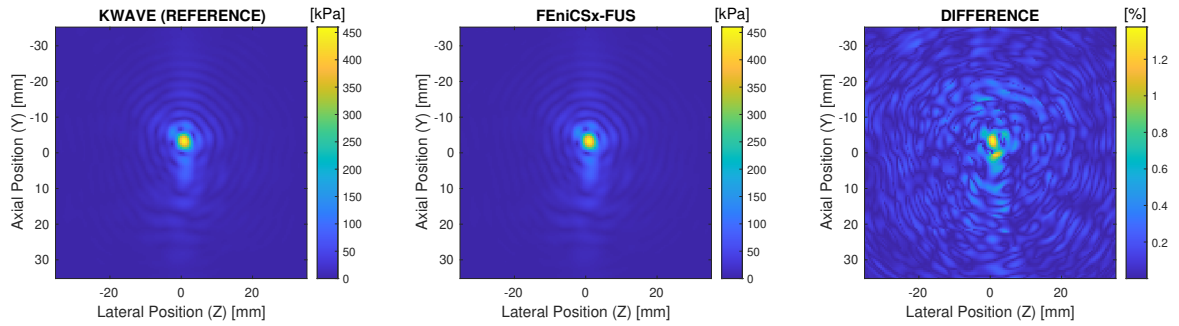
Pronounced disagreements are observed in the nearfield solution. These discrepancies arise from differences in the boundary condition models between the two solvers. As discussed in section 2.3, Neumann and absorbing boundary conditions are used to model the transducer's source. This is fundamentally different than k-Wave source's model, which uses free-field monopoles. Additionally, while k-Wave is frequently regarded as the ground truth for therapeutic ultrasound simulations, its status as such remains an open question. In fact, the benchmark study in [16] was conducted to establish a framework for model cross-comparisons. The study reported similar discrepancies between k-Wave and other therapeutic ultrasound solvers. Notably, the cross-comparison yields a median difference of less than 10% in the focal pressure between different solvers.



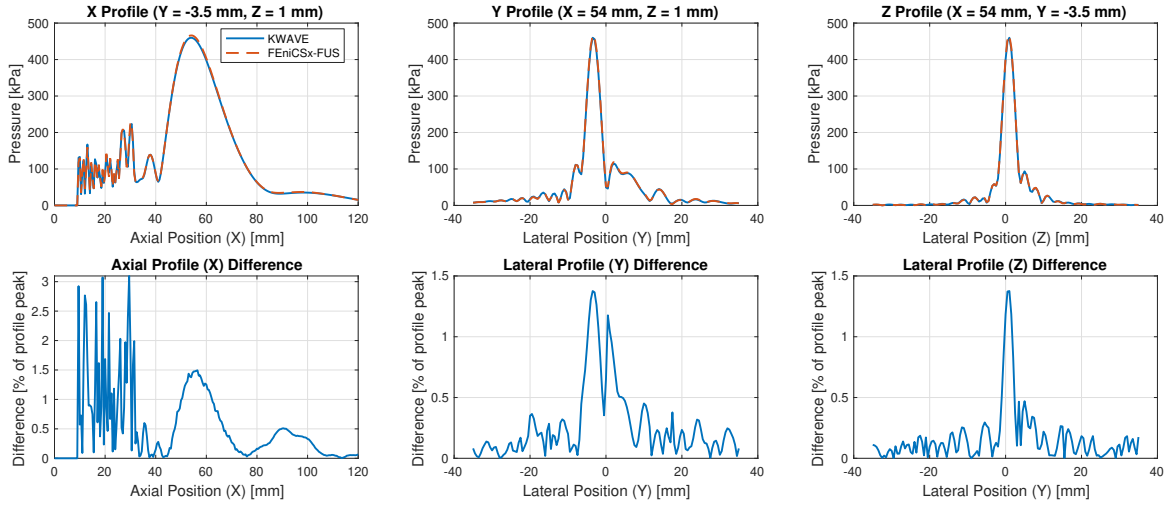
(a) The pressure amplitudes on the XY-plane for benchmark 7 with focused bowl source.



(b) The pressure amplitudes on the XZ-plane for benchmark 7 with focused bowl source.



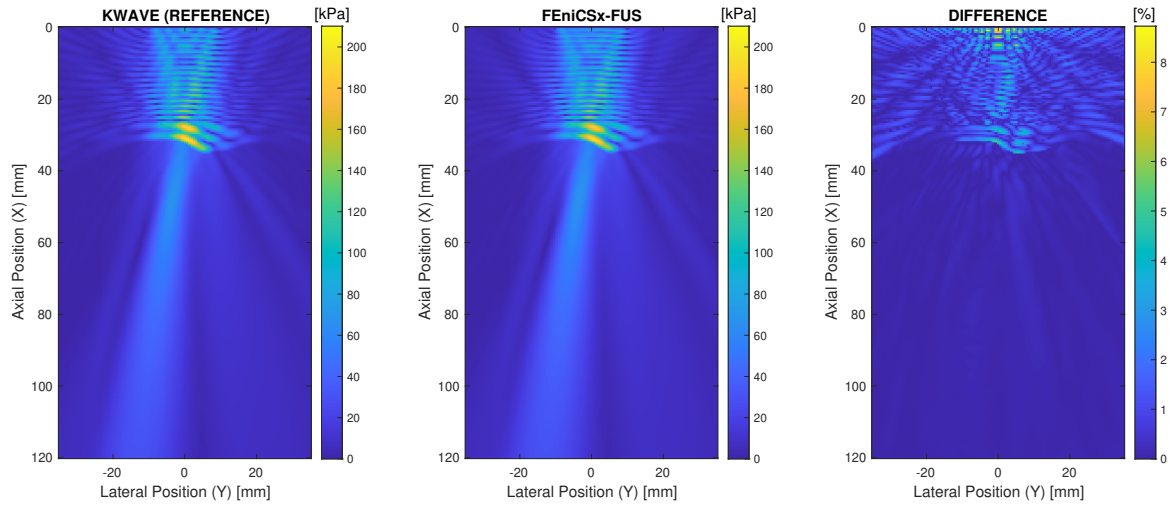
(c) The pressure amplitudes on the YZ-plane for benchmark 7 with focused bowl source.



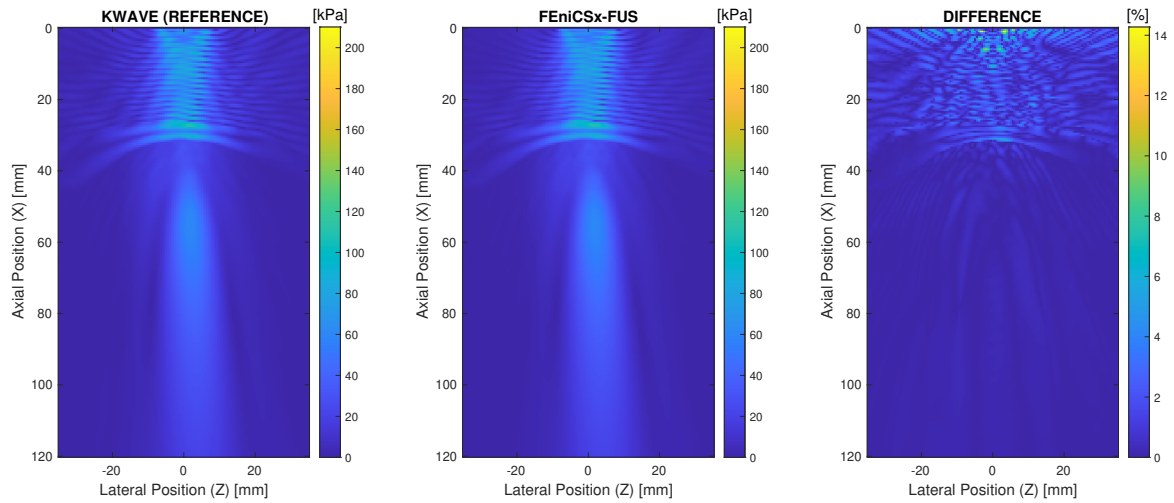
(d) The profiles for benchmark 7 with focused bowl source.

Fig. 6.26 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 7 with focused bowl source. The comparison domain size is $120 \times 70 \times 70$ mm with 0.5 mm grid sampling, which corresponds to $241 \times 141 \times 141$ grid points.

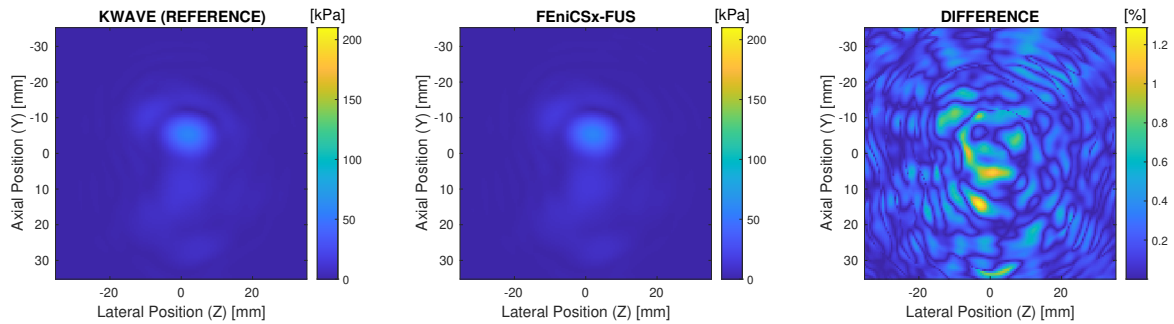
Overall, these discrepancies can be attributed to the differences in the underlying model equations, the numerical methods, the source modelling, and the boundary condition implementations.



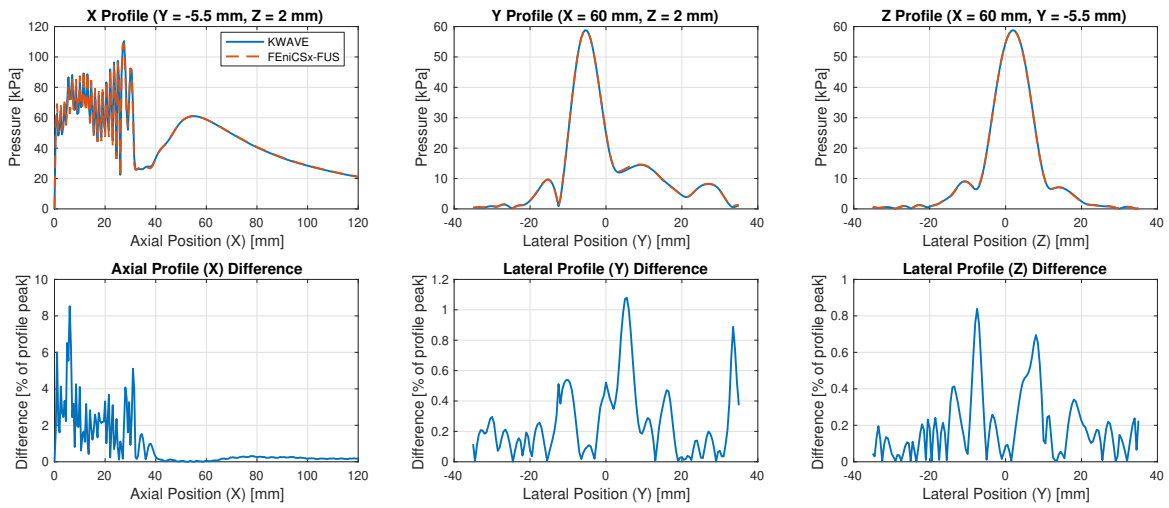
(a) The pressure amplitudes on the XY-plane for benchmark 7 with planar source.



(b) The pressure amplitudes on the XZ-plane for benchmark 7 with planar source.



(c) The pressure amplitudes on the YZ-plane for benchmark 7 with planar source.



(d) The profiles for benchmark 7 with planar source.

Fig. 6.27 The comparison between the k-Wave solver with the FEniCSx-FUS solver for benchmark 7 with planar source. The comparison domain size is $120 \times 70 \times 70$ mm with 0.5 mm grid sampling, which corresponds to $241 \times 141 \times 141$ grid points.

6.3 High-intensity focused ultrasound

In section 6.2, the FEniCSx-FUS solver was validated using the benchmark problems of transcranial focused ultrasound applications (tFUS). In tFUS, using the 2nd order linear wave equation is typically sufficient, because tFUS operates in the low-intensity regime. However, in HIFU applications, the magnitude of the ultrasound waves is high enough that the wave propagation is no longer linear, such as in thermal ablation and histotripsy applications [190, 162]. In HIFU, a widely used model equation is the Westervelt equation, which adds a nonlinear term to the linear 2nd order wave equation. This section presents the validation of the FEniCSx-FUS solver for HIFU applications using several numerical experiments.

6.3.1 Nonlinear wave propagation in 1D

To verify the correctness of the FEniCSx-FUS solver for nonlinear problems, the propagation of a single-frequency plane wave is compared to the analytical solution of the Burgers equation. The Burgers equation reads

$$\frac{\partial p}{\partial x} - \frac{\delta}{2c_0^3} \frac{\partial^2 p}{\partial \tau^2} = \frac{\beta}{\rho_0 c_0^3} p \frac{\partial p}{\partial \tau}, \quad (6.8)$$

where $\tau = t - x/c_0$ is the retarded time. There exists an analytical solution to the Burgers equation for the propagation of a single-frequency plane wave, which can be obtained from [74, 115]. An implementation of the analytical solution implemented by the k-Wave library is utilised [160].

For single-frequency plane wave propagation problem, the location of shock formation can be computed using

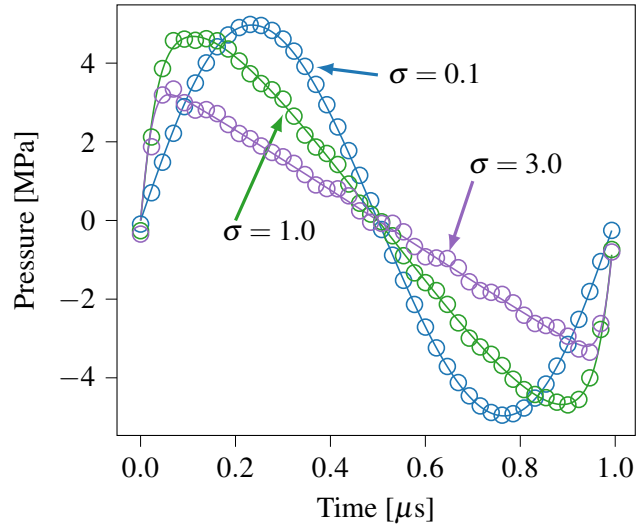
$$x_{\text{sh}} = \frac{\rho_0 c_0^3}{\beta p_0 (2\pi f_0)}, \quad (6.9)$$

where p_0 is the source pressure amplitude and f_0 is the source frequency. Furthermore, a non-dimensional parameter σ is defined such that

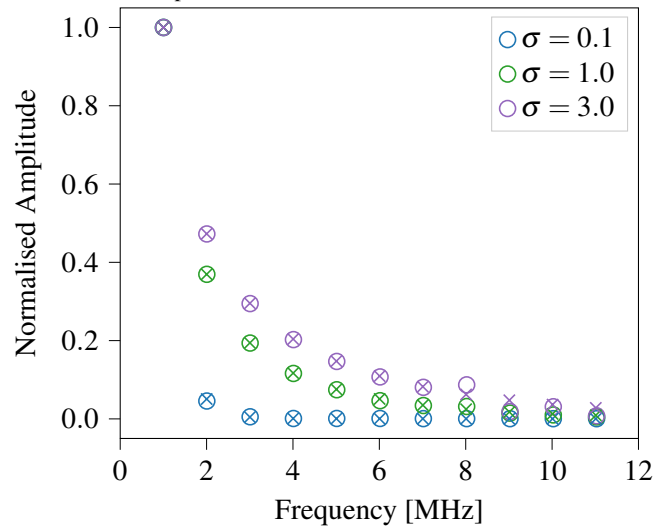
$$\sigma = \frac{x_s}{x_{\text{sh}}}, \quad (6.10)$$

where x_s is the sensor location, i.e., the parameter represents the sensor location relative to the shock location. Accordingly, $\sigma < 1.0$ represents the region before shock formation and $\sigma > 1.0$ represents the region after shock formation. In this experiment, the parameter values are as follows: $c_0 = 1500 \text{ m s}^{-1}$, $\rho_0 = 1000 \text{ kg m}^{-3}$, $\delta = 0$, $\beta = 4.8$, $f_0 = 1 \text{ MHz}$, and $p_0 = 5 \text{ MPa}$, which result in the shock formation distance, $x_{\text{sh}} = 22 \text{ mm}$.

Figure 6.28a compares the FEniCSx-FUS solution with the analytical solution for the Burgers equation of equation (6.9). The comparison was performed for a single period at three different locations, namely the location before shock formation ($\sigma = 0.1$), at the shock formation ($\sigma = 1.0$), and after shock formation ($\sigma = 3.0$). The FEniCSx-FUS solver uses the 6th degree polynomial basis function with a mesh size of 6 elements per wavelength, corresponding to 37 nodes per wavelength.



(a) Temporal comparison at three different sensor locations ($\sigma = 0.1$, $\sigma = 1.0$, $\sigma = 3.0$). The solid lines represent the analytical solution and the circle markers represent the finite element solution.



(b) Harmonic amplitudes of the finite element solution at three different sensor locations ($\sigma = 0.1$, $\sigma = 1.0$, $\sigma = 3.0$). The circle and cross marker denote the finite element and analytical solution, respectively.

Fig. 6.28 The comparison between the analytical solution and the finite element solution for the Burgers equation using a 6th degree basis function with mesh size of 6 elements per wavelength. This study is inspired by the numerical experiment performed in [163].

In contrast to a linear problem, a 6th degree basis was chosen to resolve the higher harmonics of nonlinear problems. In the region before the shock formation ($\sigma < 1.0$), the numerical solution was able to accurately capture the wave pattern. As the wave propagates toward the shock formation region, i.e., $\sigma = 1$, the wave pattern steepens, and the harmonics of higher frequencies are generated. This introduces error into the numerical solution due to the aliasing effect known as spectral blocking [163]. The aliasing effect occurs when there is leakage of spectral energies between different harmonics. As illustrated in figure 6.28b, this is apparent in $\sigma = 3$ where the spectral energy of the 7th harmonic leaks into the 8th harmonic, which leads to a higher amplitude for the 8th harmonic. In the finite element method, one approach to reduce spectral blocking is by increasing the order of the basis function, as shown in figure 6.29. The spectral blocking effects can also be minimised by increasing the number of elements per wavelength.

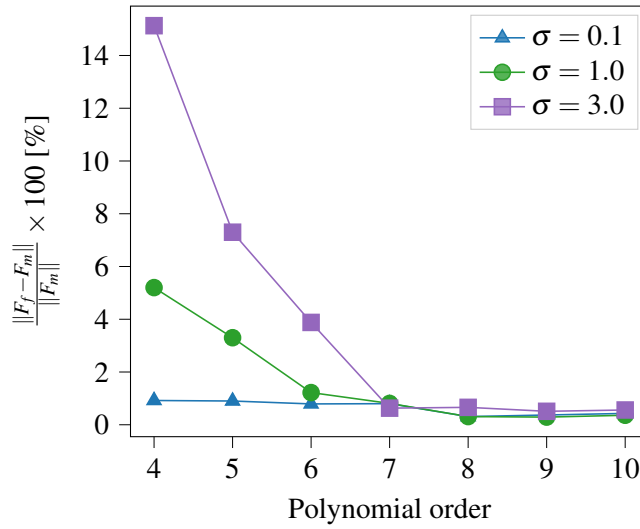


Fig. 6.29 Least square error in percentage for different polynomial order of basis function. The least square error is computed using the harmonic amplitudes, up to the 12th harmonics. The symbols F_m and F_f denote the harmonics based on analytical solution and the finite element solution, respectively.

Overall, the accuracy of the FEniCSx-FUS solver for HIFU applications depends on several factors. Most importantly, the number of elements per wavelength and the order of the basis function determine the number of harmonics that can be accurately represented. Hence, the criteria for the number of elements per wavelength and the order of the basis function depend on the particular HIFU applications.

6.3.2 Simulation of nonlinear focused ultrasound application in 3D

The FEniCSx-FUS solver is validated for 3D weakly nonlinear acoustic simulation representing a HIFU application. Specifically, a qualitative comparison to the simulation in [67] is presented. In [67], they developed a volume integral equation solver specifically for focused ultrasound application called the VINES solver [66]. The paper solved a frequency domain formulation of the Westervelt equation,

whereby the Westervelt equation is replaced by a cascade of inhomogeneous Helmholtz equation. The paper then compared their numerical solution against a solution based on the wide-angle KZK solver, namely the HITU simulator [147, 148]. A qualitative comparison is performed against both VINES and HITU simulator due to the inherent differences in the models used by each solver.

Model equation

The model equation used for the 3D nonlinear problems is the Westervelt equation, which is restated here

$$\frac{1}{\rho_0} \nabla^2 p - \frac{1}{\rho_0 c_0^2} \frac{\partial^2 p}{\partial t^2} + \frac{\delta}{\rho_0 c_0^2} \nabla^2 \frac{\partial p}{\partial t} = -\frac{\beta}{\rho_0^2 c_0^4} \frac{\partial^2 p^2}{\partial t^2}, \quad (6.11)$$

where p is the pressure field. The medium parameters c_0 , ρ_0 , δ , and β denote the speed of sound, density, diffusivity of sound, and nonlinearity coefficient, respectively. The medium parameters can be spatially varying.

Transducer configuration

As in [67], two focused bowl transducer types are considered, namely the H101 transducer and the H131 transducer [4]. The power outputs are set to 100W, a setting that is typical for thermal ablation [103, 135]. The operating frequency of both transducers is set to run at 1.1 MHz. As in section 6.2, the focused bowl transducer is modelled as a single-element transducer. A table of the transducer characteristics is given in table 6.3.

Table 6.3 The operating frequencies and geometrical data of the focused bowl transducers, taken from [4]. The geometrical parameters are the focal length l and the transducer radius R . Table reproduced from [67].

	f_0 (MHz)	l (mm)	R (mm)
H101	1.1	63.2	32
H131	1.1	35	16.5

Material parameters

To ensure comparable set-ups, an attempt is made to replicate the material parameters used in [67]. However, the FEniCSx-FUS solver is only capable of handling non-dispersive absorption, whereby the attenuation term is modelled using a frequency squared dependence such that

$$\alpha = \alpha_0 |\omega_0|^\eta, \eta = 2. \quad (6.12)$$

The symbols α_0 and η are medium specific attenuation parameters and ω_0 is the angular frequency of the source. For biological tissues, the parameter η is typically in the range of $1 \leq \eta \leq 2$ [67]. However, the model in this work is limited to $\eta = 2$. Accordingly, this may result in differences

between the FEniCSx-FUS output with the reference solvers. The material parameters used for the simulations are given in table 6.4.

Table 6.4 Material parameters for used in the numerical experiments. Obtained from [17, 51].

	ρ_0 (kg m ⁻³)	c_0 (m s ⁻¹)	β	α_0 (dB cm ⁻¹ at 1 MHz)
Water	1000	1480	3.5	0.2
Liver	1060	1590	4.4	90.0

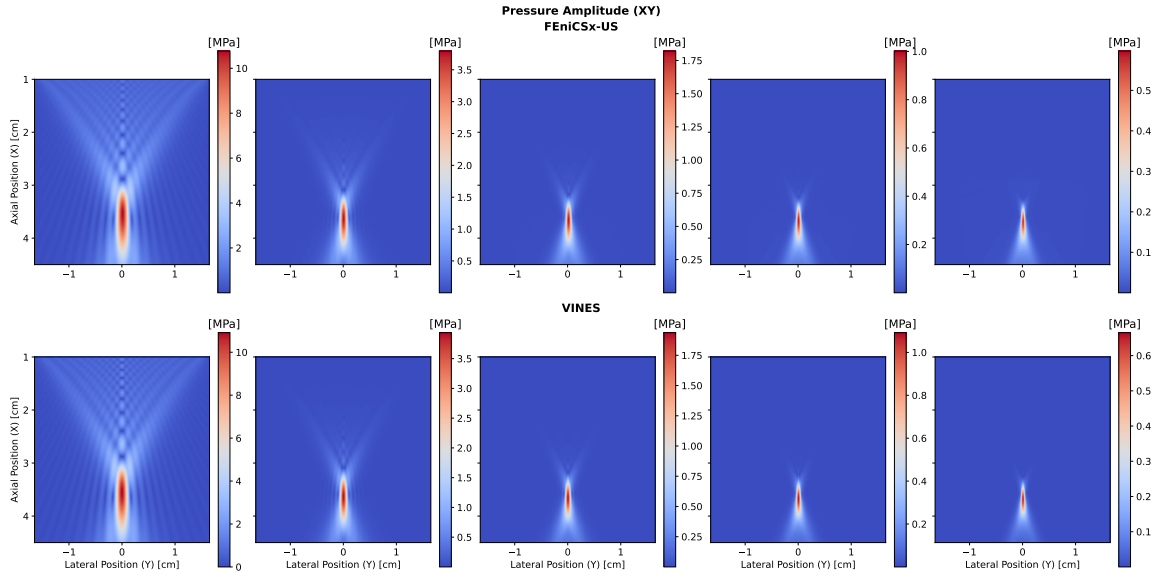
Simulation setup

Similar to section 6.2, the meshes in this section were generated using the open-source software Gmsh [62] and is available in this repository. This provides a level of control over the mesh sizes, which is important for nonlinear acoustic simulation in 3D. In contrast to the benchmark problems in section 6.2, the formation of shocks and the generation of higher harmonics necessitate a higher-order basis function and an increased in the number of elements per wavelength. Indeed, these requirements incur a higher computational cost, thus resulting in a longer simulation time. Hence, to achieve an acceptable simulation time, the elements at the focal region are made denser in contrast to other regions. It is found that 6th order basis is sufficient to obtain a close qualitative comparison with the reference solvers up to the 5th harmonics.

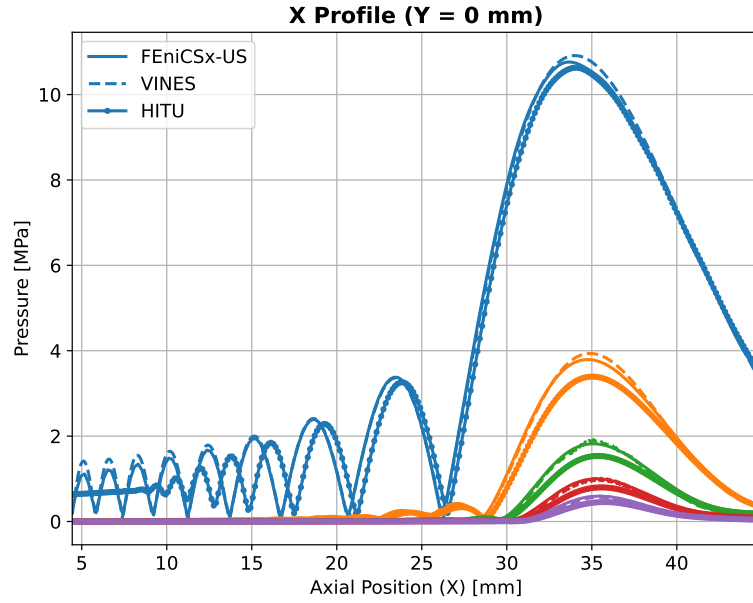
H131 transducer in water

Figure 6.30 compares the pressure amplitudes and profiles for the H131 transducer operating at 100 W in water. In figure 6.30a, close qualitative agreements are observed in the total pressure field between the FEniCSx-FUS solver with the VINES solver. In figure 6.30b, good agreements are also observed with the VINES solver across the entire domain. In contrast, the HITU solver shows large difference starting from the source surface to the focal region. This can be explained by the different in the model equation considered by each solver. In particular, the FEniCSx-FUS and VINES solver solve the Westervelt equation, whereas the HITU solver solves the wide-angle KZK equation. Accordingly, it can be seen that the HITU solver is inaccurate near the transducer surface due to the parabolic approximation made in the derivation of the WAKZK equation. Figure 6.30b shows small differences in the pressure amplitude between FEniCSx-FUS and VINES solvers, especially at the focal region. This may be due to the modelling of the source function. As previously mentioned, FEniCSx-FUS models the source function as a single surface, whereas the VINES solver considers a collection of point sources. Further, FEniCSx-FUS solver computes an approximation of the transducer surface velocity based on the power output of the source.

The FEniCSx-FUS simulation was performed on the Intel Skylake CPU using 448 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately



(a) The pressure amplitudes on the XY-plane comparison with VINES.



(b) The profiles comparison with the VINES and HITU simulator.

Fig. 6.30 The comparison between the FEniCSx-FUS solver with the reference solvers for H131 transducer in water. The comparison domain size is 80×40 mm 2D slice with 0.2 mm grid sampling, which corresponds to 357×179 grid points.

430×10^6 . The simulation took around 3 hours 20 minutes (34300 time steps), which includes saving a 2D slice of a single period to file.

H101 transducer in liver

Figure 6.31 shows the pressure amplitudes and profiles for the H101 transducer operating at 100W in the liver. A good qualitative agreement is observed between the FEniCSx-FUS solver with the VINES solver, especially in terms of the total pressure field patterns, which is shown in figure 6.31a. A slight difference can be seen in the amplitude of the 1st harmonic between the FEniCSx-FUS solver and the VINES solver. This is due to the inherent difference in modelling the attenuation term between the two solvers. The VINES solver can handle arbitrary absorption of the power law while the FEniCSx-FUS solver can only handle square dependence. While it is possible to incorporate arbitrary absorption of the power law for time-domain solver as illustrated in [161], it is currently beyond the scope of this thesis. Nonetheless, at higher harmonics, good agreements were shown between the two solvers.

The FEniCSx-FUS simulation was performed on the Intel Skylake CPU using 448 processes with single precision floating-point number. The degrees-of-freedom of the simulation is approximately 430×10^6 . The simulation took around 4 hours 43 minutes (48500 time steps), which includes saving a 2D slice of a single period to file.

6.3.3 Summary for HIFU simulations

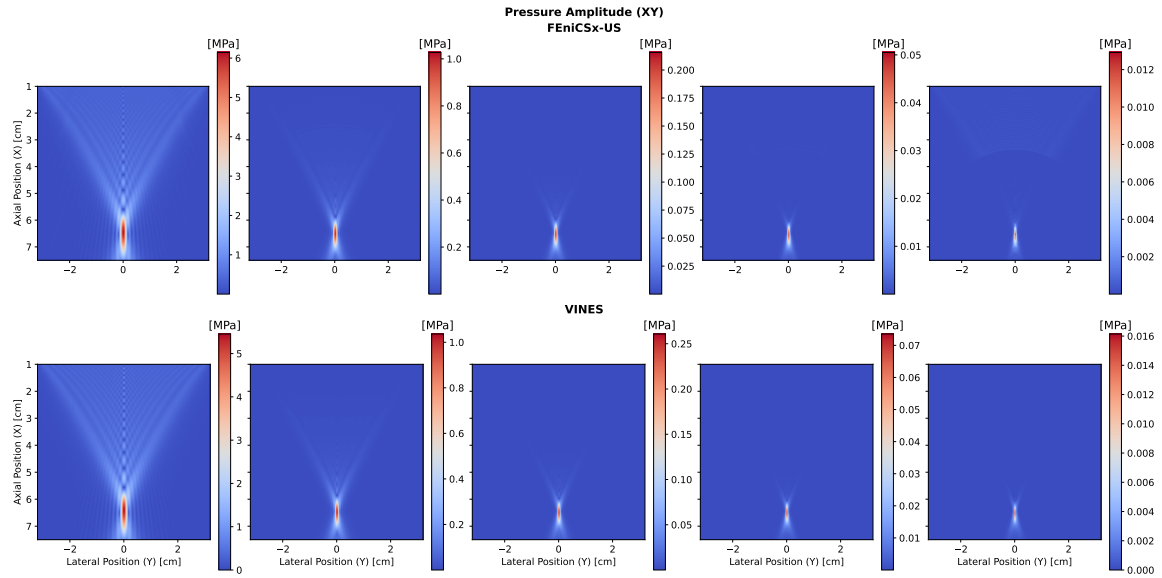
The numerical experiments validate that the FEniCSx-FUS solver is suitable for nonlinear acoustic simulations. In particular, the solver is capable of handling HIFU simulations, which extends the capability of the FEniCSx-FUS solver to handle nonlinear scenarios. Furthermore, good qualitative agreements were found between the FEniCSx-FUS and the reference solvers (VINES and HITU).

While VINES and HITU are useful for qualitative comparisons with FEniCSx-FUS, a quantitative study is essential to validate FEniCSx-FUS for nonlinear problems. In particular, the VINES solver is currently limited to homogeneous media and depends on nested meshes [67], which may not be suitable for realistic FUS simulations. Future work will focus on a quantitative comparison between FEniCSx-FUS and other nonlinear ultrasound solvers, such as k-Wave.

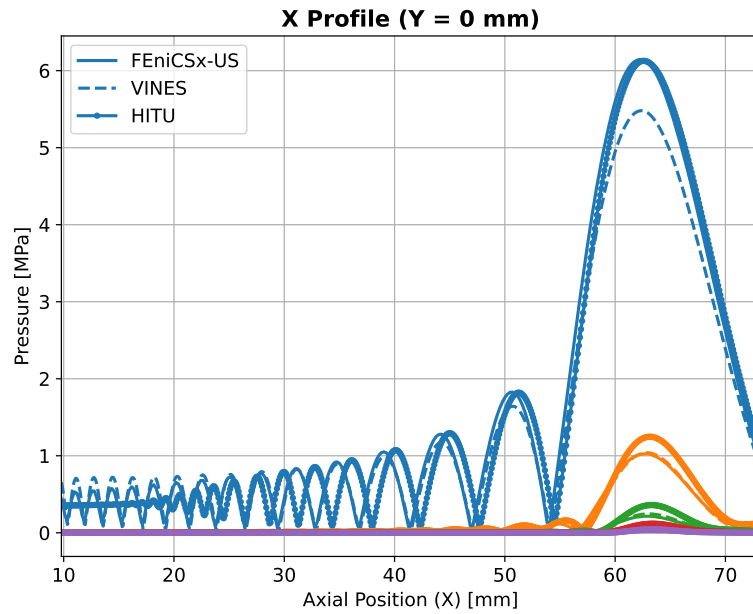
Overall, similar to section 6.2, the differences observed between the solvers can be attributed to the differences in the numerical methods, the source modelling, and the boundary condition implementations.

6.4 Summary

In section 6.2, the simulation times for the benchmark problems were found to range from minutes to hours. The simulation time is restricted by the number of the total time steps of the problem, which in turns depends on the time step size. As the time step size is determined by equation (6.5), it is thus problem dependent. However, it is remarked here that the meshes in this benchmark problems were



(a) The pressure amplitudes on the XY-plane comparison with VINES.



(b) The profiles comparison with the VINES and HITU simulator.

Fig. 6.31 The comparison between the FEniCSx-FUS solver with the reference solvers for H101 transducer in liver. The comparison domain size is 120×64 mm 2D slice with 0.2 mm grid sampling, which corresponds to 536×287 grid points.

generated *manually*, which may be suboptimal with regard to its effect on the time step size. Although, the subject of mesh generation is beyond the scope of this thesis and is set for future research.

Another important finding is that the saving of a 3D volume is currently quite costly as can be seen from benchmark 7 of section 6.2. Even though the number of degrees-of-freedom and time steps is relatively small, the simulation time took more than 5 hours. This result suggests that a majority of the solve time is spent on outputting the solution to file. Indeed, this result is unsurprising since a *naive* data output process was implemented, whereby the solution is appended to a text file. Hence, a future project for the FEniCSx-FUS solver is to speed up the writing of the FEniCSx-FUS solution to file, especially for 3D outputs.

In section 6.3, longer simulation times were observed for the HIFU simulation as compared to the tFUS simulation in section 6.2. This is unsurprising due to several factors. Firstly, the HIFU model is nonlinear, thus, it requires additional simulation data in contrast to the tFUS model. Secondly, to resolve the nonlinear wave pattern requires a higher order basis function and more number of elements per wavelength, which translates to a larger number of degrees-of-freedom as opposed to the tFUS model.

Chapter 7

GPU accelerated solver

In this chapter, the FEniCSx-FUS solver is extended for GPU architectures. The parallelism that is inherent in the finite element assembly process naturally lend itself to GPU acceleration. In view of this, a GPU version of the FEniCSx-FUS solver is implemented using the Python Numba library [105]. The Numba library is utilised because of its built-in capability to implement CUDA kernels. Also, a Python implementation is considered due to its ease of implementation as compared to C++. Despite this, it will be shown that a Python GPU implementation gives comparable performance to a C++ CPU implementation.

In section 7.1, the differences between a CPU and GPU architecture are briefly discussed. In section 7.2, key concepts of a CUDA implementation are presented, focussing in particular on how the GPU kernel is setup and called. Building on these concepts, in section 7.3, the GPU implementation of the operator kernels is presented, namely the mass and stiffness operator kernels. In section 7.4, the MPI communicators that enable multi-GPU utilisation are presented, i.e., the scatter forward and scatter reverse communicators. Next, in section 7.5, the kernels are examined to ensure that the kernels achieve a good fraction of peak performance. Additionally, the MPI communication performance is shown to achieve near perfect strong scaling. Finally, in section 7.6, the outputs of the GPU implementation are validated against the CPU implementation. In this regard, the GPU implementation is validated using both a linear and a nonlinear 3D example, hence, verifying that the GPU implementation is correct.

7.1 Difference between a CPU and GPU architecture

GPUs are highly specialised hardware designed for parallel computations. CPUs and GPUs are designed with different goals [5]. CPUs are designed to excel at controlling the logical flow of any general purpose program. In contrast, GPUs are designed for highly parallel computations where more transistors are devoted to data processing instead of control flow. As a result, programs involving a high number of floating-point computations benefit from the use of GPUs, such as in graphics rendering and finite element integration of high degree basis function [100]. Figure 7.1 shows a

schematic comparison of CPU and GPU architectures. Note that the number of cores (green boxes) on a GPU is significantly higher than on a CPU.

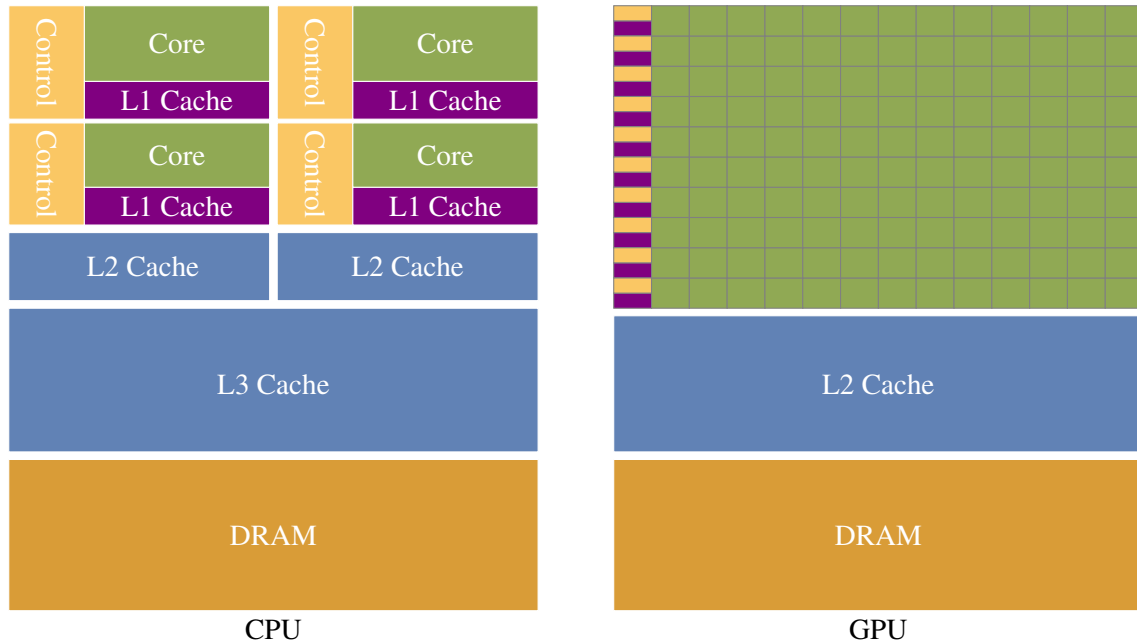


Fig. 7.1 A schematic comparison between CPU and GPU architectures. Schematic reproduced from [5].

7.2 Basics of a CUDA kernel implementation

A key distinction between the CPU and GPU implementations lies in the control that the developer has on the threads and blocks of a GPU architecture. In particular, a GPU kernel allows the developer to assign specific tasks to specific threads and blocks. This functionality is provided by the GPU kernel where the developer is able to access a specific thread or block based on its index. As an illustrative example, consider a simplified CUDA¹ kernel provided below, which performs a copy operation between two arrays on the *device*²

```
@cuda.jit
def copy(a, b):
    idx = cuda.threadIdx.x + cuda.blockDim.x * cuda.blockIdx.x
    if idx < a.size:
        b[i] = a[i]
```

Note the `idx` variable in the above kernel: it defines the thread index. To ensure that each element in array `a` is copied to array `b`, the appropriate number of threads and blocks needs to be assigned. For

¹an API that is developed by NVIDIA to program kernels for NVIDIA GPUs.

²a term whereby the array is in the GPU memory. Its complement is the *host*, a term whereby the array is in the CPU memory.

instance, a common choice for the number of threads per block is 128. Therefore, given an array of size N , the number of blocks is set to be

$$N_b = (N + (N_t - 1)) \% N_t \quad (7.1)$$

where the subscript b and t are blocks and threads per block, respectively. The symbol $\%$ denotes the floor division operator, ensuring that the number of blocks is rounded down to the nearest integer. Since each thread operates independently, the kernel can execute in parallel, thus accelerating the process. In general, a particular thread index can be accessed based on the built-in `threadIdx`, `blockDim`, and `blockIdx` special objects provided by CUDA. Furthermore, CUDA provides `x`, `y`, and `z` attributes for each of these objects.

7.3 Operator kernels

In this section, the GPU implementations of the operator kernels are presented. Specifically, the CUDA implementations of the mass operator kernel of equation (5.4) and the stiffness operator kernel of equation (5.7) are presented. The key differences between the GPU implementation and the CPU implementation are highlighted.

7.3.1 Mass operator kernel

The mass operator is implemented with the idea that the number of blocks and number of threads per block is set according to the number of cells and number of DOF per cell, respectively. The mass operator of equation (5.4) reads

$$m_q^k = u_q w_q |J_k(\hat{x}_q)|. \quad (7.2)$$

The assembly of the mass operator involves two indices, namely the k index that represents the cell and the q index that represents the quadrature points. Thus, the assembly of the mass operator kernel is an *embarrassingly parallel*³ operation and is suited for GPU implementation. The index k is assigned to the number of cells and the index q is assigned to the number of threads. Since the mass lumped finite element scheme is used, the number of DOF per cell is equal to the number of quadrature points.

Listing 7.1 shows a snippet of the implementation of the mass operator kernel using CUDA. Line 1–3 shows how the global thread index is extracted based on the block and thread indices. Given the global thread index, the entity and local DOF are then extracted as shown in lines 5 and 6. Once the entity and local DOF are obtained, the global DOF is extracted as shown in line 10. Notice that in the GPU implementation, in contrast to the CPU implementation, there are no loops over the entities (e.g., cells) and the local DOF. Indeed, the entities and local DOF are indexed using the built-in thread and block indices, respectively. This enables the kernel to operate on the global DOF in parallel and

³a workload that can be easily divided into independent tasks, which can be performed in parallel. Typically, these tasks involve little to no communication or dependencies between them.

ensures that a single thread handles a single DOF per entity. Additionally, the implementation uses an atomic operation, namely the atomic add function (line 16). This operation is necessary to prevent a race condition, in particular, when the contributions of the local DOF are added to the output array. It is remarked here that the mass operator implementation can handle both cell and facet entities.

```

1 thread_id = cuda.threadIdx.x # Local thread ID (max: 1024)
2 block_id = cuda.blockIdx.x # Block ID (max: 2147483647)
3 idx = thread_id + block_id * cuda.blockDim.x # Global thread ID
4
5 entity = idx // entity_dofmap.shape[1]
6 local_dof = idx % entity_dofmap.shape[1]
7
8 if idx < entity_dofmap.size:
9     # Compute the global DOF index
10    dof = entity_dofmap[entity, local_dof]
11
12    # Compute the contribution of the current DOF to the mass operator
13    value = x[dof] * detJ_entity[entity, local_dof] * entity_constants[entity]
14
15    # Atomically add the computed value to the output array 'y'
16    cuda.atomic.add(y, dof, value)

```

Listing 7.1 Numba CUDA mass operator.

7.3.2 Stiffness operator kernel

In contrast to the mass operator kernel, the stiffness operator kernel requires the tensor contraction process as illustrated in section 5.3. Algorithm 2 shows the steps in the stiffness operator that uses the sum-factorisation algorithm. In section 5.3, the CPU implementation of algorithm 2 was shown.

Algorithm 2 Stiffness operator.

Input: $\mathbf{x}$, $\mathbf{G}$, DOFMAP, Φ

Output: $\mathbf{y}$

- 1: Extract the local DOF, $\mathbf{x}_k = \mathbf{x}[\text{DOFMAP}]$.
 - 2: Apply contraction in the x , y , and z -direction based on equation (5.31), $\mathbf{x}_c = \Phi \mathbf{x}_k$.
 - 3: Apply the transformation, $\mathbf{x}_t = \mathbf{G} \mathbf{x}_c$.
 - 4: Apply contraction in the x , y , and z -direction based on equation (5.31), $\mathbf{y}_k = \Phi^T \mathbf{x}_t$.
 - 5: Add the contributions, $\mathbf{y}[\text{DOFMAP}] = \mathbf{y}_k$.
-

Listing 7.2 shows the implementation of algorithm 2 using CUDA. Like the mass operator kernel, the stiffness operator kernel does not loop over entities or local DOF. However, instead of indexing using a single thread index, the stiffness operator uses the built-in x , y , and z thread indices. This enables the kernel to treat the local DOF in the cell entity naturally as a tensor. Observe that the kernel

does not required any transposition process of the local DOF. Note that the loops in the kernel are equivalent to the tensor contraction operation.

Since each coordinate direction is independent in the tensor contraction, the process can be parallelised. However, care must be taken to avoid race conditions, which would lead to incorrect results due to threads reading or writing from the same memory location. Race conditions are avoided by using `cuda.syncthreads`, as illustrated in listing 7.2. Additionally, to allow threads to access data from different threads, shared arrays are necessary. This is accomplished using the `cuda.shared.array` function. Using shared arrays is also faster since the array is stored in shared memory. It is important to highlight that the `float32` function is required to prevent a change in the floating-point precision. In particular, the CUDA kernel defaults to `float64` if a variable is initialised within the kernel. Similar to listing 7.1, the atomic add operation also avoids race conditions.

```

1  tx = cuda.threadIdx.x
2  ty = cuda.threadIdx.y
3  tz = cuda.threadIdx.z
4
5  thread_id = tx * cuda.blockDim.y * cuda.blockDim.z + ty * cuda.blockDim.z +
    tz
6  block_id = cuda.blockIdx.x
7
8  # Get dof index that this thread is computing
9  dof = entity_dofmap[block_id, thread_id]
10
11 scratch = cuda.shared.array(shape=(n, n, n), dtype=float_type)
12 scratchx = cuda.shared.array(shape=(n, n, n), dtype=float_type)
13 scratchy = cuda.shared.array(shape=(n, n, n), dtype=float_type)
14 scratchz = cuda.shared.array(shape=(n, n, n), dtype=float_type)
15
16 # Gather x expression value required by this thread
17 scratch[tx, ty, tz] = x[dof]
18 cuda.syncthreads()
19
20 # Apply contraction in the x-direction
21 val_x = float32(0.0)
22 for ix in range(n):
23     val_x += dphi[tx, ix] * scratch[ix, ty, tz]
24
25 # Apply contraction in the y-direction
26 val_y = float32(0.0)
27 for iy in range(n):
28     val_y += dphi[ty, iy] * scratch[tx, iy, tz]
29
30 # Apply contraction in the z-direction
31 val_z = float32(0.0)
32 for iz in range(n):
33     val_z += dphi[tz, iz] * scratch[tx, ty, iz]

```

```

34
35 # Apply transform
36 G0 = G_entity[block_id, thread_id, 0]
37 G1 = G_entity[block_id, thread_id, 1]
38 G2 = G_entity[block_id, thread_id, 2]
39 G3 = G_entity[block_id, thread_id, 3]
40 G4 = G_entity[block_id, thread_id, 4]
41 G5 = G_entity[block_id, thread_id, 5]
42 coeff = entity_constants[block_id]
43
44 fw0 = coeff * (G0 * val_x + G1 * val_y + G2 * val_z)
45 fw1 = coeff * (G1 * val_x + G3 * val_y + G4 * val_z)
46 fw2 = coeff * (G2 * val_x + G4 * val_y + G5 * val_z)
47
48 scratchx[tx, ty, tz] = fw0
49 scratchy[tx, ty, tz] = fw1
50 scratchz[tx, ty, tz] = fw2
51
52 cuda.syncthreads()
53 # Apply contraction in the x-direction
54 val_x = float32(0.0)
55 for ix in range(n):
56     val_x += dphi[ix, tx] * scratchx[ix, ty, tz]
57
58 # Apply contraction in the y-direction
59 val_y = float32(0.0)
60 for iy in range(n):
61     val_y += dphi[iy, ty] * scratchy[tx, iy, tz]
62
63 # Apply contraction in the z-direction
64 val_z = float32(0.0)
65 for iz in range(n):
66     val_z += dphi[iz, tz] * scratchz[tx, ty, iz]
67
68 # Add contributions
69 val = val_x + val_y + val_z
70
71 # Atomically add the computed value to the output array 'y'
72 cuda.atomic.add(y, dof, val)

```

Listing 7.2 Numba CUDA stiffness operator.

7.4 Multi-GPU implementation

To effectively utilise a multi-GPU system, an implementation of the scatter forward and scatter reverse functions that are capable of transferring data between GPUs without passing through the CPU are

required. The implementation is inspired by the scatter forward and scatter reverse functions in DOLFINx. In fact, the implementation is equivalent to the concepts presented in section 4.2.

As discussed in section 4.2, the implementation of a scatterer requires information about the *ghost* DOF and their owning processes. Specifically, it is necessary to identify which processes own the ghost DOF contained within a particular process. DOLFINx provides this information through the IndexMap object. Firstly, the ghost DOF in a particular process that are owned by other processes are obtained. This is accomplished as follows.

```

1  imap = V.dofmap.index_map
2  owners = imap.owners
3  unique_owners, owners_size = np.unique(owners, return_counts=True)
4  owners_argsorted = np.argsort(owners)
5
6  owners_offsets = np.cumsum(owners_size)
7  owners_offsets = np.insert(owners_offsets, 0, 0)
8
9  owners_idx = [np.zeros(size, dtype=np.int64) for size in owners_size]
10 for i, owner in enumerate(unique_owners):
11     begin = owners_offsets[i]
12     end = owners_offsets[i + 1]
13     owners_idx[i] = owners_argsorted[begin:end]
14
15 owners_idx_d = [cuda.to_device(owner_idx) for owner_idx in owners_idx]

```

Listing 7.3 Obtaining and sorting the ghosts degrees-of-freedom within a process.

Listing 7.3 shows the steps to obtain and sort the set of ghost DOF within a process before sending it to the owning processes. The ghost DOF in a DOLFINx array are placed at the end of the array. However, the ordering of the owning process for the ghost DOF are random. Thus, to ensure that the correct ghost DOF are sent to the owning process, the ghost DOF are sorted into ascending order by the MPI process. This is accomplished in lines 1-4. Next, a list containing arrays of the ghost DOF is created for each owning processes. Finally, the arrays are copied from the *host* to the *device*. This is accomplished in lines 9-15.

It is vital that each process knows which of the DOF that it owns are ghosts on other processes. This information is obtained as follows

```

1  nlocal = imap.size_local
2  shared_dofs = imap.index_to_dest_ranks()
3  shared_ranks = np.unique(shared_dofs.array)
4
5  ghosts = []
6  for shared_rank in shared_ranks:
7      for dof in range(nlocal):
8          if shared_rank in shared_dofs.links(dof):
9              ghosts.append(shared_rank)
10
11 ghosts = np.array(ghosts)

```

```

12 unique_ghosts, ghosts_size = np.unique(ghosts, return_counts=True)
13 ghosts_offsets = np.cumsum(ghosts_size)
14 ghosts_offsets = np.insert(ghosts_offsets, 0, 0)
15
16 all_requests = []
17
18 # Send
19 send_buff_idx = [np.zeros(size, dtype=np.int64) for size in owners_size]
20 for i, owner in enumerate(unique_owners):
21     begin = owners_offsets[i]
22     end = owners_offsets[i + 1]
23     send_buff_idx[i] = imap.ghosts[owners_argsorted[begin:end]]
24     reqs = comm.Isend(send_buff_idx[i], dest=owner)
25     all_requests.append(reqs)
26
27 # Receive
28 recv_buff_idx = [np.zeros(size, dtype=np.int64) for size in ghosts_size]
29 for i, ghost in enumerate(unique_ghosts):
30     reqr = comm.Irecv(recv_buff_idx[i], source=ghost)
31     all_requests.append(reqr)
32
33 MPI.Request.Waitall(all_requests)
34
35 ghosts_idx = [recv_buff - imap.local_range[0] for recv_buff in recv_buff_idx]
36
37 ghosts_idx_d = [cuda.to_device(ghost_buff) for ghost_buff in ghosts_idx]

```

Listing 7.4 Obtaining the owning degrees-of-freedom in a process that are ghosts in other processes.

Listing 7.4 shows steps that identify DOF on a process that are ghosts on other processes. These DOF are determined by checking whether the DOF is a shared DOF. This information is crucial in order to allocate the size of the arrays to receive from processes containing the ghost DOF. Accordingly, this is accomplished in lines 1-14. Next, the MPI communication steps are executed, whereby the global numbering of the ghost DOF is sent to the owning processes. This is accomplished in lines 16-33. Subsequently, in line 35, the local numbering of the shared DOF is obtained. Finally, the arrays are copied from the *host* to the *device* in line 37. The steps to obtain `owners_idx_d` and `ghosts_idx_d` can be done on the *host* since it is computed only once and reused by the scatterer.

7.4.1 Scatter forward implementation

The scatter forward function sends the DOF from the owning process to the ghost DOF on other processes. Listing 7.5 shows the scatter forward implementation for the GPU solver. The scatter forward function begins by packing the DOF from the buffer—an array containing the *local* and *ghost* DOF. More specifically, the packing kernel involves the extraction of the DOF from the buffer into the `send_buff`. The variable `send_buff` is a list containing *device* arrays such that each array corresponds to each shared process. Next, the MPI communication step is executed, whereby each

array in the `send_buff` is sent to its corresponding processes. Finally, the DOF are unpacked from the `recv_buff` into the buffer. The `pack_fwd` and `unpack_fwd` functions are implemented as CUDA kernels and are illustrated in listings A.1 and A.2, respectively.

```

1 def scatter_forward(buffer: numba.types.Array):
2     """
3     Perform the scatter forward operation of the buffer array.
4
5     Parameters
6     -----
7     buffer : array to perform scatter forward
8     """
9
10    all_requests = []
11
12    # Set the number of threads
13    threadsperblock = 128
14
15    # Pack
16    numblocks_pack = [
17        (ghost_size + (threadsperblock - 1)) // threadsperblock
18        for ghost_size in ghosts_size
19    ]
20    for i, sb in enumerate(send_buff):
21        pack_fwd[numblocks_pack[i], threadsperblock](buffer, sb, ghosts_idx[i])
22
23    # Synchronize
24    cuda.synchronize()
25
26    # Send
27    for i, dest in enumerate(ghosts):
28        reqs = comm.Isend(send_buff[i], dest=dest)
29        all_requests.append(reqs)
30
31    # Receive
32    for i, src in enumerate(owners):
33        reqr = comm.Irecv(recv_buff[i], source=src)
34        all_requests.append(reqr)
35
36    MPI.Request.Waitall(all_requests)
37
38    # Unpack
39    numblocks_unpack = [
40        (owner_size + (threadsperblock - 1)) // threadsperblock
41        for owner_size in owners_size
42    ]
43    for i, rb in enumerate(recv_buff):

```

```

44     unpack_fwd[numblocks_unpack[i], threadsperblock](
45         rb, buffer, owners_idx[i], N
46     )
47
48     # Synchronize
49     cuda.synchronize()

```

Listing 7.5 Scatter forward kernel.

7.4.2 Scatter reverse implementation

The scatter reverse kernel sends the ghost DOF to the the owning process. Listing 7.6 shows the scatter reverse implementation for the GPU solver. The scatter reverse function begins by packing the DOF from the buffer, particularly, the packing kernel involves the extraction of the ghost DOF from the buffer to be stored into `send_buff`. Equivalent to the scatter forward kernel, the variable `send_buff` is a list containing *device* arrays, whereby each array contains ghost DOF for each shared process. Next, the MPI communication step is performed, such that each *device* arrays are send to its owning process. Finally, the DOF are unpacked from the `recv_buff` into the buffer. The `pack_rev` and `unpack_rev` functions are implemented as CUDA kernels and are given in listings A.3 and A.4, respectively.

```

1  def scatter_reverse(buffer: numba.types.Array):
2      """
3      Perform the scatter reverse operation of the buffer array.
4
5      Parameters
6      -----
7      buffer : array to perform scatter reverse
8      """
9
10     all_requests = []
11
12     # Set the number of threads
13     threadsperblock = 128
14
15     # Pack
16     numblocks_pack = [
17         (owner_size + (threadsperblock - 1)) // threadsperblock
18         for owner_size in owners_size
19     ]
20     for i, sb in enumerate(send_buff):
21         pack_rev[numblocks_pack[i], threadsperblock](buffer, sb, owners_idx[i], N)
22
23     # Synchronize
24     cuda.synchronize()

```

```

25
26     # Send
27     for i, dest in enumerate(owners):
28         reqs = comm.Isend(send_buff[i], dest=dest)
29         all_requests.append(reqs)
30
31     # Receive
32     for i, src in enumerate(ghosts):
33         reqr = comm.Irecv(recv_buff[i], source=src)
34         all_requests.append(reqr)
35
36     MPI.Request.Waitall(all_requests)
37
38     # Unpack
39     numblocks_unpack = [
40         (ghost_size + (threadspblock - 1)) // threadspblock
41         for ghost_size in ghosts_size
42     ]
43     for i, rb in enumerate(recv_buff):
44         unpack_rev[numblocks_unpack[i], threadspblock](rb, buffer,
45             ghosts_idx[i])
46
47     # Synchronize
48     cuda.synchronize()

```

Listing 7.6 Scatter reverse kernel.

7.5 Performance analysis

A critical step in an effective GPU implementation is profiling its performance. In this section, the performance of the GPU implementation is measured and validated using Nsight Systems [7] and Nsight Compute [6] provided by NVIDIA. The goal is to analyse the performance of the operator kernels and scatterers. Nsight Systems provides a comprehensive performance overview of the solver. On the other hand, Nsight Compute provides a more detailed performance measure such as providing a roofline analysis of a kernel. Furthermore, Nsight Compute is able to identify the specific line of code that may affect the kernel performance. However, profiling using Nsight Compute typically requires administrator access, which makes examining the kernel performance more difficult especially on GPU clusters.

7.5.1 Operator kernels performance

The performance analysis of the operators follows the same idea from section 4.6, i.e., by comparing the performance of each kernels, namely the mass kernels and the stiffness kernels. To this end, the kernels are profiled using a single NVIDIA RTX A4000 GPU [3]. The GPU consists of 16GB of

memory with a theoretical memory bandwidth of 448GB s^{-1} . Its single precision performance is 19.2TFLOPS .

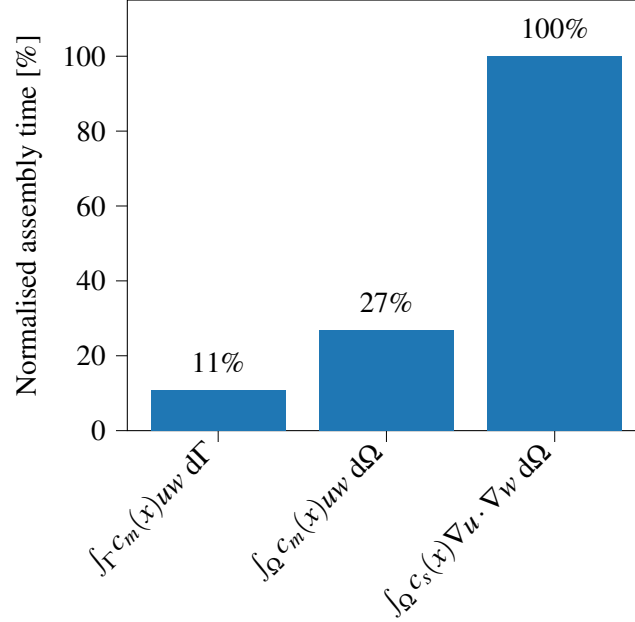


Fig. 7.2 The normalised assembly times for the operator kernels. The symbols Γ and Ω denote integration over facets and cells, respectively. This was profiled on the NVIDIA RTX A4000 GPU with 2.1×10^6 degrees-of-freedom using a $P4$ continuous Lagrange finite element. The absolute time for each operator are $0.16 \times 10^{-3}\text{s}$ for the mass operator on the boundary, $0.40 \times 10^{-3}\text{s}$ for the mass operator on the cell, and $1.5 \times 10^{-3}\text{s}$ for the stiffness operator on the cell.

Figure 7.2 shows the normalised assembly times for the kernels. In particular, it shows the percentage of time with respect to the most expensive kernel, namely the stiffness operator kernel. Similar to the CPU solver, the main bottleneck of the GPU solver is the stiffness kernel. This result was expected since the stiffness kernel contains more floating-point operations than the mass kernel. Motivated by this result, the stiffness kernel performance on the GPU is examined.

Figure 7.3 shows the roofline plot of the stiffness kernel obtained from NVIDIA Nsight Compute. The roofline plot records the performance in FLOPS per second for the 64-bit and 32-bit implementations of the stiffness kernel. Observe that the 64-bit floating-point precision (circle with red edge) achieves its maximum performance, that is about 0.14TFLOPS s^{-1} . Further, observe that the implementation is compute-bound for the 64-bit floating-point precision. In contrast, the 32-bit floating-point precision (circle with blue edge) implementation is memory-bound. Unlike the 64-bit floating-point precision, the 32-bit implementation does not achieve its peak performance.

The observations in figure 7.3 warrant further study. A comparison to assess the performance impact of the degrees-of-freedom map ordering is performed. Figure 7.4 shows the performance between

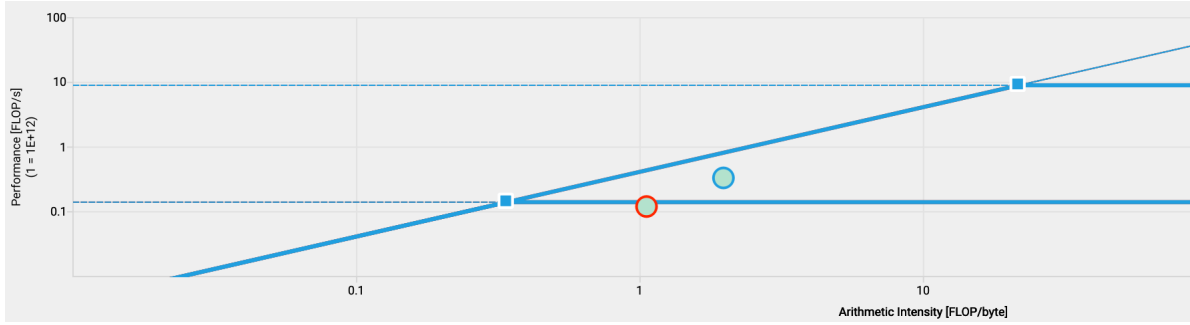


Fig. 7.3 The roofline plot for the stiffness kernel between the 64-bit (circle with red edge) and 32-bit (circle with blue edge) floating-point precision. This data was collected by running an experiment with 33×10^6 degrees-of-freedom using a $P4$ continuous Lagrange finite element. The experiment was performed on a single NVIDIA RTX A4000 GPU.

the Basix⁴ and tensor product ordering of the degrees-of-freedom map. Indeed, the tensor product ordering gives a better performance and achieves close to the theoretical maximum performance. Furthermore, the tensor product ordering reduces the arithmetic intensity of the stiffness kernel. These improvements stem from the better memory access pattern of the tensor product ordering. In fact, in this experiment, the memory throughput of the tensor product ordering increases by 75% relative to the Basix ordering.

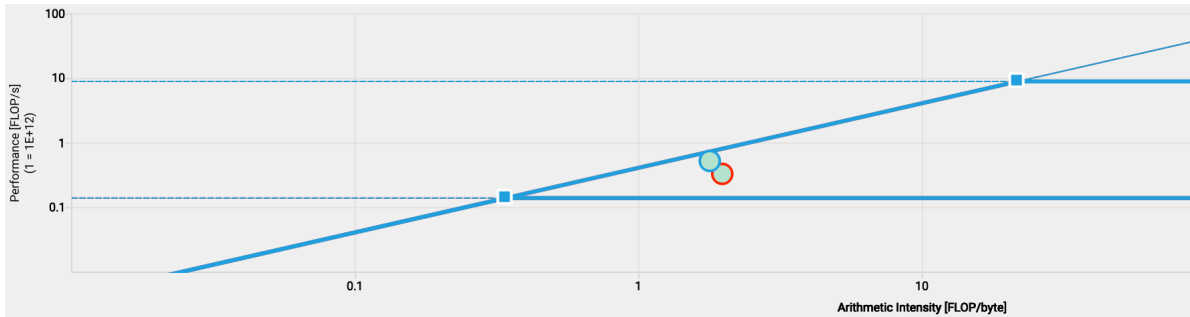


Fig. 7.4 The roofline plot for the stiffness kernel between the Basix (circle with red edge) and tensor product (circle with blue edge) order. This data was collected by running an experiment with 33×10^6 degrees-of-freedom using $P4$ continuous Lagrange finite element. The experiment was performed on a single NVIDIA RTX A4000 GPU.

A novel feature of the solver is the ease of using a higher order polynomial basis function. Accordingly, a study was performed to analyse the performance of the stiffness kernel across several polynomial orders, i.e., from $P3$ to $P7$. Figure 7.5 shows the roofline plot of the stiffness kernel for different polynomial orders. It is observed that for the element order $P3$ to $P6$, the stiffness kernel performed quite well. In fact, for these elements, the kernel is utilising greater than 80% of the available performance of the device, which explains the excellent performance of the GPU solver.

⁴Basix ordering is the convention used by Basix to order the degrees-of-freedom of the finite element basis. Basix orders the degrees-of-freedom as follows: vertex, edge, face, volume. This ordering is clearly different from the tensor product ordering. The convention used by Basix for degrees-of-freedom ordering can be obtained from [140].

However, a marked drop in performance is noticeable for $P7$. This outcome is surprising since a $P7$ element is expected to perform well because the number of threads per block is 8. This expectation is based on warp efficiency whereby when the number of threads per block is a multiple of 32, it ensures that warps are fully utilised, hence, avoids wasted resources and maximises efficiency. However, an explanation for this result has not yet been established and has been listed as future research.

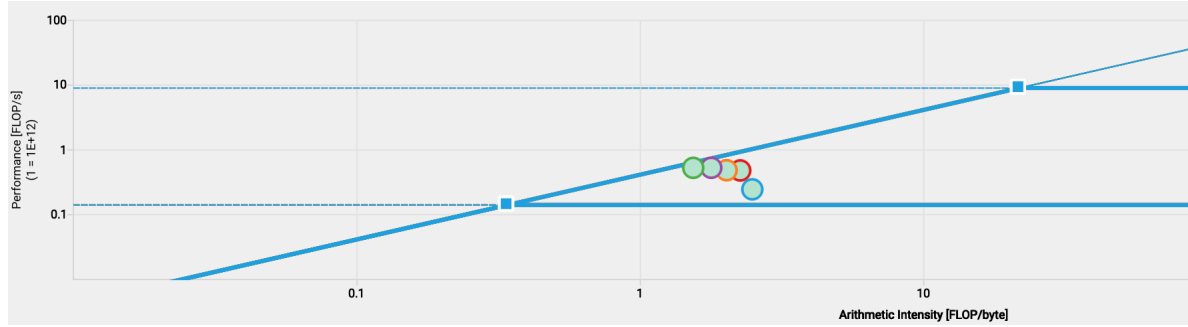


Fig. 7.5 The roofline plot for the stiffness kernel for different polynomial order. The edge color of the circle represents different polynomial order. In increasing order, the green edge represent $P3$, the violet edge represent $P4$, the orange edge represent $P5$, the red edge represent $P6$, and the blue edge represent $P7$. The basis function is of tensor product ordering. The experiment was performed on a single NVIDIA RTX A4000 GPU.

7.5.2 Multi-GPU performance

A key objective in the implementation of a scatterer is the efficient transfer of data between *devices* without going through the *host*. This is important because data movement between the *device* and *host* is a recognised bottleneck common in GPU implementations. The Nsight Systems profiler can be used to examine the data transfer between *devices*. Figure 7.6 shows the Nsight Systems report for the scatter forward and scatter reverse GPU implementations. Indeed, the Nsight Systems report shows a peer-to-peer memory transfer, i.e., *device-to-device* communications, which validates that data is transferred between *devices*, thereby achieving the implementation objective.

In addition, a scaling experiment was performed on the GPU implementation. Figure 7.7 shows the result of a strong scaling experiment on up to 8 NVIDIA A100 GPUs with a DGX platform on the DICC machine at the University of Malaya. The results demonstrate near perfect strong scaling between 0.5 seconds per time step to 0.125 seconds per time step. The result illustrates the excellent performance of the MPI implementation on the NVIDIA DGX architecture.

7.6 Simulation examples

In this section, validations of the GPU implementation are presented. Since the numerical formulation is similar to the CPU implementation, only two examples taken from sections 6.2 and 6.3 were considered. For the linear example, benchmark 1 with planar source problem that was defined in section 6.2 was considered. Next, for the nonlinear example, the H131 transducer in water problem

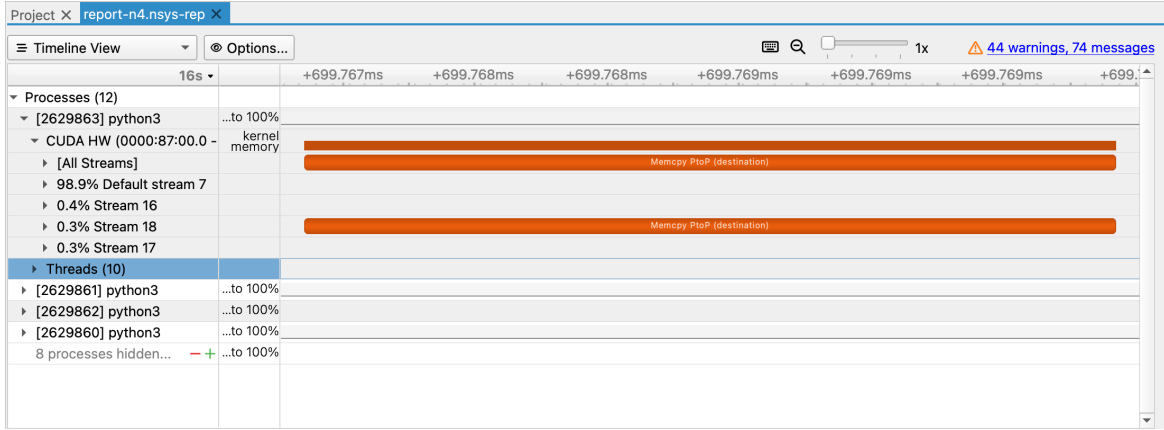


Fig. 7.6 The Nsight Systems report of the MPI communication. The Memory PtoP denotes the peer-to-peer communication between the *devices*. This experiment was performed using MPI on a connection of 4 A100 NVIDIA GPU.

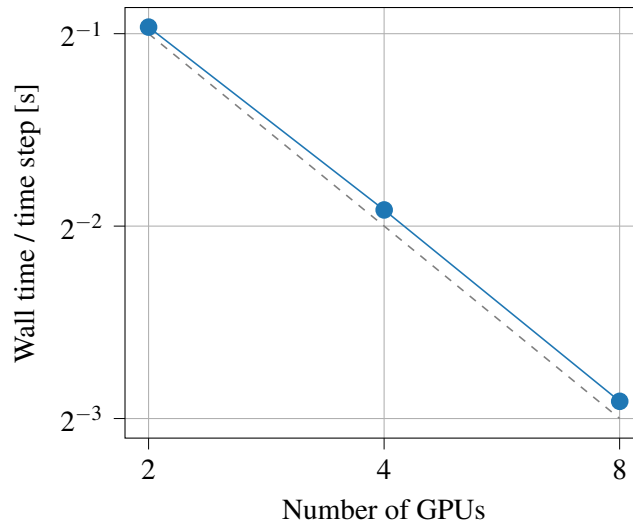
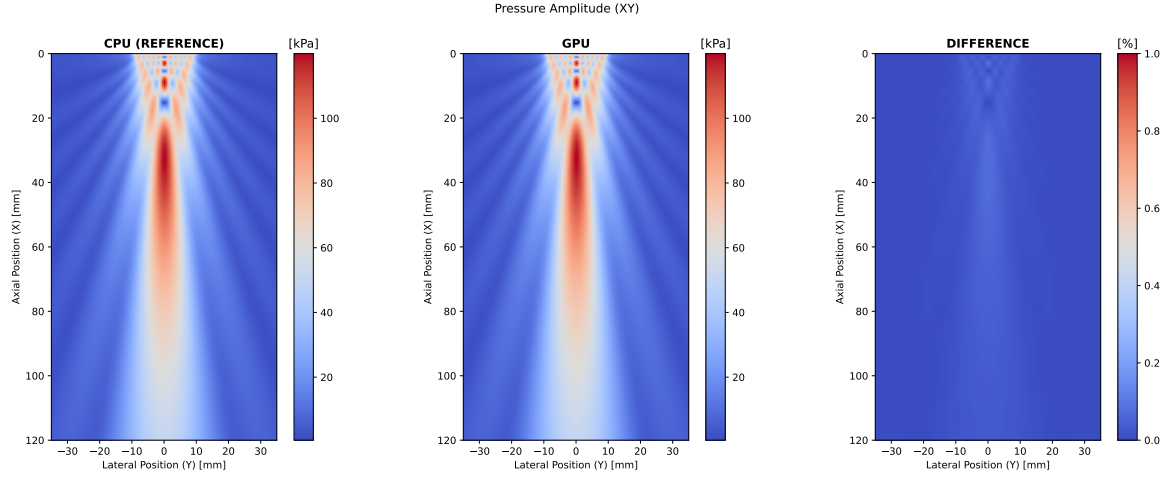


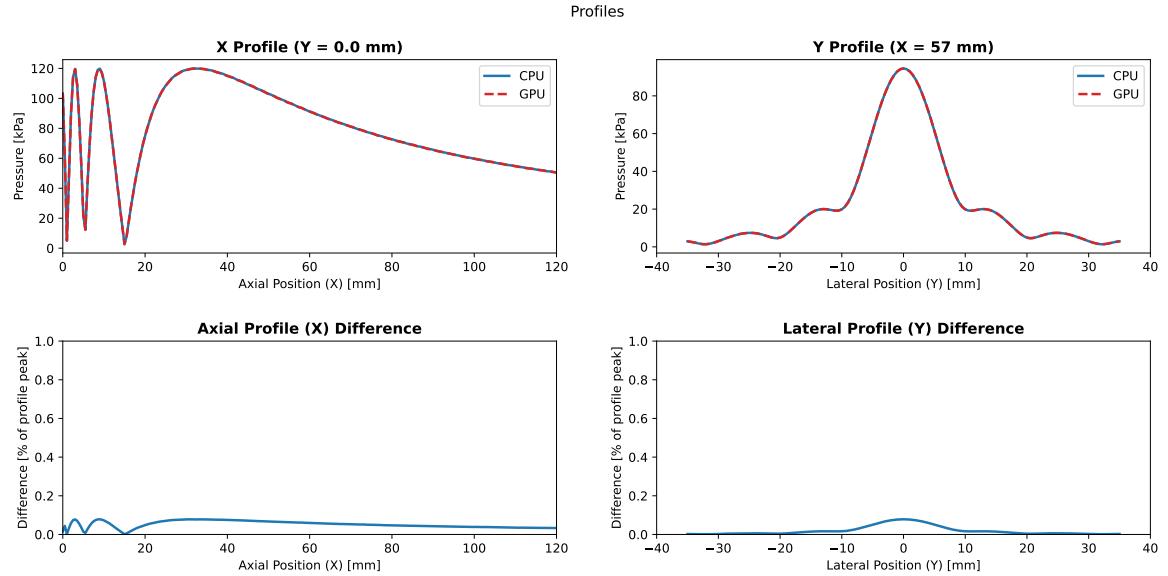
Fig. 7.7 Strong scaling analysis of the FEniCSx-FUS GPU solver on NVIDIA DGX A100 platform for problem size of 430 million degrees-of-freedom. The dashed grey line represents perfect scaling and the blue line represents scaling achieved. The experiment was performed using 4th order explicit Runge–Kutta method with 34300 time steps. The data shows the run time per step.

that was defined in section 6.3 was considered. In both cases, the GPU solutions are compared with the CPU solutions.

7.6.1 Linear example



(a) The pressure amplitudes comparison for benchmark 1 with planar source.



(b) The profiles comparison for benchmark 1 with planar source.

Fig. 7.8 The comparison between the CPU and GPU implementation of the FEniCSx-FUS solver for benchmark 1 with planar source.

Figure 7.8 presents a comparison of the pressure amplitudes and profiles for benchmark 1 between the CPU and GPU implementations of the FEniCSx-FUS solver. As expected, very small differences

are observed between the two solvers, with the maximum difference being less than 0.1%. This result is unsurprising since both implementations are built on the same numerical method.

The GPU simulation was performed on a single NVIDIA RTX A4000 GPU with double floating-point precision. The GPU simulation took around 6 minutes, which includes saving a 2D slice of a single period to file. This level of performance is comparable to a single node of a HPC cluster, which is remarkable considering that the NVIDIA RTX A4000 is a consumer-grade GPU. These findings demonstrate the potential for enabling patient-specific treatment planning, particularly for transcranial focused ultrasound treatments.

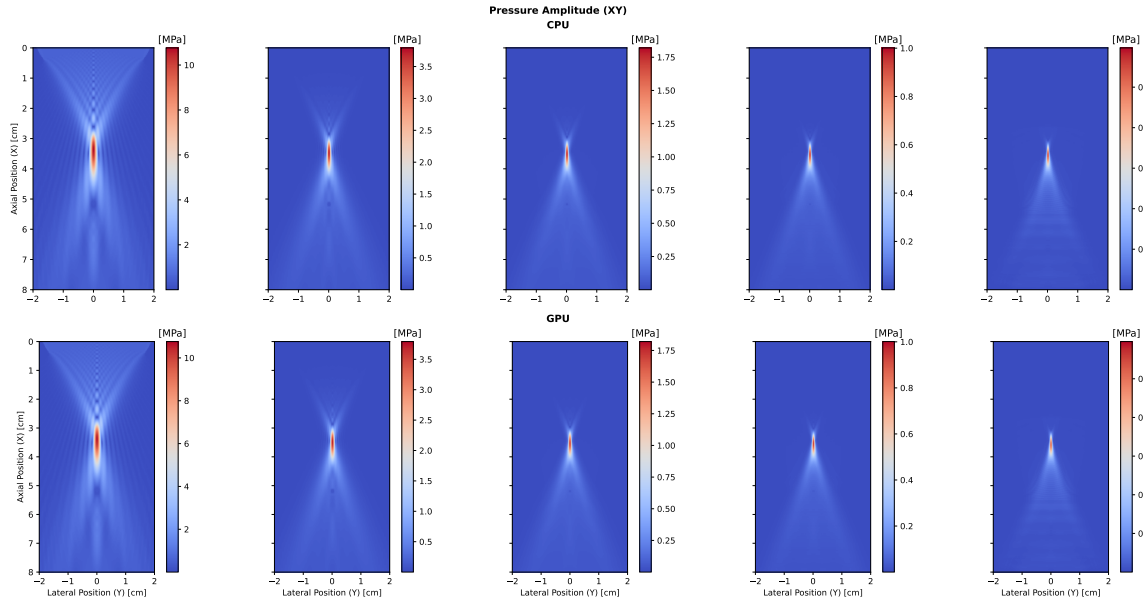
7.6.2 Nonlinear example

Figure 7.9 presents a comparison of the pressure amplitudes and profiles for H131 transducer at 100W in water between the CPU and GPU implementations of the FEniCSx-FUS solver. Like figure 7.8, there are no significant differences between the outputs of the CPU and GPU implementations.

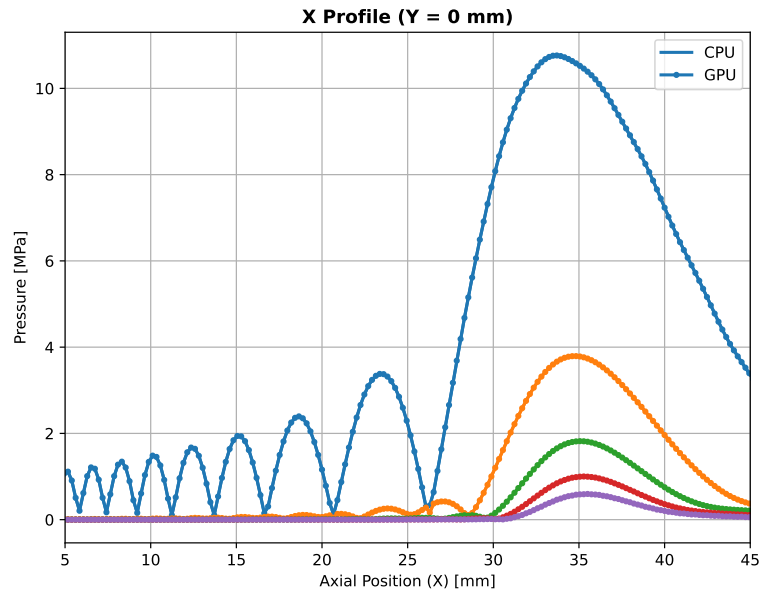
The GPU simulation was performed on two NVIDIA A100 GPUs using MPI with double floating-point precision. The GPU simulation took about 5 hours, which includes saving a 2D slice of a single period to file. This is longer than the CPU simulation, which took around 3 hours 20 minutes (see section 6.3). However, it is important to note that the CPU simulation was ran on 448 processes across 14 nodes of Skylake CPUs, whereas the GPU simulation was ran on a single node and only two GPUs. This suggests that it is possible to run a HIFU simulation without access to large computer clusters.

7.7 Summary

In this chapter, the GPU implementation of the FEniCSx-FUS solver is presented. Unlike the CPU implementation, the GPU implementation of the mass operator kernel handles facet-wise entities. The scatter forward and scatter reverse communicators were implemented, which enables multi-GPU utilisation and eliminates the reliance on DOLFINx functions for these tasks. Plus, these additional implementations ensure that the GPU solver runs entirely on the GPU instead of a mixture of CPU and GPU. Performance analyses to validate the GPU implementation were presented. In this regard, it was shown that the operator kernels performed close to the GPU theoretical peak performance. The multi-GPU data transfer was shown to move directly between *devices*. Additionally, it was shown that the GPU solver achieves good strong scaling. Validations of the GPU solver were performed against the CPU solver. Indeed, the GPU solver outputs a solution that is very close to the CPU solver.



(a) The pressure amplitudes comparison for the H131 transducer at 100W in water.



(b) The profiles comparison for the H131 transducer at 100W in water.

Fig. 7.9 The comparison between the CPU and GPU implementation of the FEniCSx-FUS solver for H131 transducer at 100W in water.

Chapter 8

Conclusion and future research

Focused ultrasound applications hold the potential for the treatment of cancers such as liver and kidney tumours and neurological disorders such as Parkinson's and Alzheimer's diseases. However, widespread adoption of FUS technology has been hindered by the challenge of prescribing patient-specific treatment planning. The issue lies in accurately predicting the effects of an FUS treatment on a patient's body within a reasonable time frame. Addressing this issue requires the development of a fast solver capable of modelling the behaviour of focused ultrasound treatments. This thesis presents the development of a fast finite element solver for focused ultrasound applications.

The development of a solver for focused ultrasound applications involves solving a mathematical model. The Westervelt equation was utilised due to its ability to accurately represent the nonlinear behaviour inherent in high-intensity focused ultrasound applications. To this end, the mathematical derivation of the Westervelt equation was presented. Additionally, to provide a complete mathematical description for a focused ultrasound simulation, the boundary and initial conditions were presented. In particular, the Sommerfeld boundary condition was imposed to ensure that the mathematical problem is physically valid.

The mathematical problem was solved numerically. The variational form of the problem was formulated. Galerkin approximation was applied to obtain a semi-discrete form of the variational problem. To construct a fully discrete form, the Runge–Kutta scheme was utilised for temporal discretisation. Specifically, the explicit 4th order Runge–Kutta scheme was applied. A high-order finite element method with mass lumping was used to reduce the pollution error. It was shown that the explicit 4th order Runge–Kutta scheme with mass lumping avoids the linear solving process, thus, reduces the computational cost. Additionally, the assembly of a matrix is avoided, which reduces the memory requirements. A discussion of the degree-of-freedom map for high-order basis was presented, which is a key concept that enables the solver to utilise arbitrary degree basis function. The combination of the mass lumped finite element method and 4th order explicit Runge–Kutta scheme constitutes the complete numerical procedure of the solver. An overview of the procedures taken by the solver to simulate a FUS application were also presented.

The finite element method was implemented with the FEniCSx open-source software using the C++ API. The rationale behind this choice is the access to low-level functions provided by the C++ API. The components that make up the FEniCSx-FUS solver were presented, namely the UFL problem description and the implementation of the Runge–Kutta time-stepping scheme. Furthermore, the technique to obtain a diagonal mass *matrix* and how to define cell-wise heterogeneity using FEniCSx were demonstrated. The parallel processing capability that is key to enabling large-scale acoustic simulation was presented. Also, the solver’s bottleneck was identified to be the cell-wise finite element assembly.

The performance of the cell-wise finite element assembly was improved by implementing fast finite element integration. Since the mesh for a FUS simulation is typically fixed during the entire simulation, the precomputation of geometric data was implemented. In particular, the scaled Jacobian determinant and the geometric factor were computed and stored. Hence, reduces the number of floating-point operations at each Runge–Kutta time step. Additionally, exploiting the tensor-product structure of the mesh, the sum-factorisation algorithm was implemented. The implementation was designed with consideration for the memory access pattern and data locality to ensure optimal performance. Furthermore, a reduced precision option was implemented for the solver. Given these implementations, tests were performed to assess the solver’s performance. Firstly, it was found that the precomputation of geometric data speed-up the finite element assembly process. Regarding the sum-factorisation implementation, it was shown to achieve above 55% of the CPU architecture theoretical peak performance, demonstrating a good fraction of peak performance. Further, it was shown that the finite element assembly achieved near perfect speed-up with 32-bit precision relative to 64-bit precision. Finally, in terms of the overall performance, the FEniCSx-FUS solver demonstrated good strong scaling behaviour.

The solver’s capability to handle a range of simulation setups was also examined, starting with the application of source windowing. It was demonstrated that windowing eliminates the generation of high-frequency waves that could pollute the finite element solution. Additionally, validation tests were conducted on the FEniCSx-FUS solver for a set of simulation setups. The solver’s capability was demonstrated using different transducer configurations, multiple heterogeneous media, nonlinearity, and realistic simulation scenarios such as in tFUS and HIFU applications. Overall, the results indicated that the FEniCSx-FUS solution closely agrees with established numerical solvers for FUS applications.

The parallelism in the FEniCSx-FUS implementation prompted the extension of the solver for GPU architecture. Accordingly, the FEniCSx-FUS solver was implemented for GPU using the Python Numba library. The GPU implementation is eased by DOLFINx data-oriented design, whereby the data required for the GPU kernel are easily obtained through the DOLFINx Python API. As a result, the finite element assembly operator was reimplemented, namely the mass and stiffness operators for GPU. Additionally, the MPI communicators—the scatterers—were implemented to enable multi-GPU utilisation. Performance analyses were performed to examine the operator kernels performance. The operator kernels were found to perform at a good fraction of peak performance. Furthermore, the MPI

communicators were shown to be capable of transferring data directly between GPUs. In terms of multi-GPU utilisation, the solver showed excellent strong scaling behaviour of up to 8 GPUs. Finally, it was demonstrated that the GPU and CPU implementation produced the same output, thus, verifying the correctness of the GPU implementation.

8.1 Future research

The research presented in this thesis shows the development of a fast finite element solver for focused ultrasound applications. There are several other avenues whereby the solver capabilities can be extended. The following are promising directions of future research:

- **Incorporating the bioheat equation.** At the moment, the FEniCSx-FUS solver is limited to predicting the pressure field for a focused ultrasound simulation. A natural extension for the solver is the incorporation of the bioheat equation to predict the level of heating due to a focused ultrasound treatment [123, 147, 145]. This feature is important for thermal ablation therapy [40, 70, 150].
- **Investigation of alternative power law absorption models.** The acoustic absorption of biological tissues differs depending on their type. As such, using a single power law absorption model to simulate heterogeneous tissue may result in an inaccurate simulation. As stated in section 2.1, the FEniCSx-FUS solver uses a quadratic power law absorption model, assuming that different tissue types have the same attenuation. Since FEniCSx-FUS is a time-domain solver, this makes the handling non-trivial in contrast to a frequency-domain solver [69]. Nonetheless, there are studies that have been found to model power law absorption models for time-domain solver [163]. Hence, in future work, the aim is to incorporate the FEniCSx-FUS solver with arbitrary frequency attenuation power laws. Given the flexibility of the finite element method, it is anticipated that the solver will be capable of handling different power law attenuation models in various material regions. Additionally, since the finite element method can naturally account for heterogeneity within each mesh cell, it is expected that even intra-tissue heterogeneity can be modelled.
- **Mixed mesh capability with mass lumping scheme.** The current FEniCSx-FUS solver design choices, such as the usage of fully tensor-product elements, aim to enable the utilisation of a nodal quadrature mass lumping scheme to enhance solver performance. However, in practice, the generation of a fully automatic, high quality hexahedral mesh remains an open problem [125]. Accordingly, adding a mixed-mesh capability feature on the FEniCSx-FUS solver would enhance the solver utility. However, to maintain the mass lumping design, the formulation of triangular and tetrahedral element for mass lumping is required [142, 33, 60].
- **Accelerating solver output for visualisation.** As demonstrated in chapter 6, a major bottleneck of the solver is to output 3D data for visualisation. At the moment, a *naive* approach is used

whereby the output are appended to a text file. This approach is expensive, especially when running the solver in many CPU processes. Thus, a future research endeavour is to accelerate the data output process of the solver.

- **Implementation of a user-friendly Python interface.** The current implementation of the FEniCSx-FUS solver requires the user to use C++. However, the use of C++ is typically less user-friendly compared to languages like Python or Matlab. To promote widespread adoption of the FEniCSx-FUS solver, it is essential to provide a more user-friendly interface, such as one in Python. This concept is not new, as many computational softwares—including FEniCSx—provide a Python user interface while maintaining a C++ backend to ensure optimal performance [24, 15, 138, 22].

References

- [1] (2008). *Bump Functions and Partitions of Unity*, pages 127–134. Springer New York, New York, NY.
- [2] (2021). FEniCSx Roadmap. <https://fenicsproject.org/roadmap/>. Accessed: 2023-11-15.
- [3] (2022). *NVIDIA RTX A4000*. NVIDIA. Datasheet.
- [4] (2022). Therapy transducers.
- [5] (2024). *CUDA C++ Programming Guide*. NVIDIA. Programming Guide.
- [6] (2024a). *NVIDIA Nsight Compute*. NVIDIA. Documentation.
- [7] (2024b). *NVIDIA Nsight Systems*. NVIDIA. User Guide.
- [8] Aanonsen, S. I., Barkve, T., Tjøtta, J. N., and Tjøtta, S. (1984). Distortion and harmonic generation in the nearfield of a finite amplitude sound beam. *The Journal of the Acoustical Society of America*, 75(3):749–768.
- [9] Abdelfattah, A., Barra, V., Beams, N., Bleile, R., Brown, J., Camier, J.-S., Carson, R., Chalmers, N., Dobrev, V., Dudouit, Y., Fischer, P., Karakus, A., Kerkemeier, S., Kolev, T., Lan, Y.-H., Merzari, E., Min, M., Phillips, M., Rathnayake, T., Rieben, R., Stitt, T., Tomboulides, A., Tomov, S., Tomov, V., Vargas, A., Warburton, T., and Weiss, K. (2021). GPU algorithms for Efficient Exascale Discretizations. *Parallel Computing*, 108:102841.
- [10] Afanasiev, M., Boehm, C., van Driel, M., Krischer, L., Rietmann, M., May, D. A., Knepley, M. G., and Fichtner, A. (2019). Modular and flexible spectral-element waveform modelling in two and three dimensions. *Geophysical Journal International*, 216(3):1675–1692.
- [11] Africa, P., Salvador, M., Gervasio, P., Dede’, L., and Quarteroni, A. (2023). A matrix-free high-order solver for the numerical solution of cardiac electrophysiology. *Journal of Computational Physics*, 478:111984.
- [12] Ainsworth, M. and Wajid, H. A. (2009). Dispersive and Dissipative Behavior of the Spectral Element Method. *SIAM Journal on Numerical Analysis*, 47(5):3910–3937.
- [13] Alison Burgess, Kairavi Shah, O. H. and Hynynen, K. (2015). Focused ultrasound-mediated drug delivery through the blood–brain barrier. *Expert Review of Neurotherapeutics*, 15(5):477–491.
- [14] Alnæs, M. S., Logg, A., Ølgaard, K. B., Rognes, M. E., and Wells, G. N. (2014). Unified Form Language: A Domain-Specific Language for Weak Formulations of Partial Differential Equations. *ACM Trans. Math. Softw.*, 40(2).

- [15] Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.-S., Cervený, J., Dobrev, V., Dudouit, Y., Fisher, A., Kolev, T., Pazner, W., Stowell, M., Tomov, V., Akkerman, I., Dahm, J., Medina, D., and Zampini, S. (2021). MFEM: A Modular Finite Element Methods Library. *Computers & Mathematics with Applications*, 81:42–74.
- [16] Aubry, J.-F., Bates, O., Boehm, C., Butts Pauly, K., Christensen, D., Cueto, C., G  lat, P., Guasch, L., Jaros, J., Jing, Y., Jones, R., Li, N., Marty, P., Montanaro, H., Neufeld, E., Pichardo, S., Pinton, G., Pulkkinen, A., Stanzola, A., Thielscher, A., Treeby, B., and van't Wout, E. (2022). Benchmark problems for transcranial ultrasound simulation: Intercomparison of compressional wave models. *The Journal of the Acoustical Society of America*, 152(2):1003–1019.
- [17] Azhari, H. (2010). *Appendix A: Typical Acoustic Properties of Tissues*, pages 313–314. John Wiley & Sons, Ltd.
- [18] Babu  ka, I., Ihlenburg, F., Paik, E. T., and Sauter, S. A. (1995). A Generalized Finite Element Method for solving the Helmholtz equation in two dimensions with minimal pollution. *Computer Methods in Applied Mechanics and Engineering*, 128(3):325–359.
- [19] Baker, A. C., Berg, A. M., Sahin, A., and Tj  tta, J. N. (1995). The nonlinear pressure field of plane, rectangular apertures: Experimental and theoretical results. *The Journal of the Acoustical Society of America*, 97(6):3510–3517.
- [20] Baratta, I. A., Dean, J. P., Dokken, J. S., Habera, M., Hale, J. S., Richardson, C. N., Rognes, M. E., Scroggs, M. W., Sime, N., and Wells, G. N. (2023). DOLFINx: The next generation FEniCS problem solving environment.
- [21] Berenger, J. P. (1994). A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114:185–200.
- [22] Betcke, T. and Scroggs, M. W. (2021). Bempp-cl: A fast Python based just-in-time compiling boundary element library. *Journal of Open Source Software*, 6(59):2879.
- [23] Betcke, T., van 't Wout, E., and G  lat, P. (2017). *Computationally Efficient Boundary Element Methods for High-Frequency Helmholtz Problems in Unbounded Domains*, pages 215–243. Springer International Publishing, Cham.
- [24] Blatt, M., Burchardt, A., Dedner, A., Engwer, C., Fahlke, J., Flemisch, B., Gersbacher, C., Gr  ser, C., Gruber, F., Gr  ninger, C., Kempf, D., Kl  f  korn, R., Malkmus, T., M  thing, S., Nolte, M., Piatkowski, M., and Sander, O. (2016). The Distributed and Unified Numerics Environment, Version 2.4. *Archive of Numerical Software*, 4(100):13–29.
- [25] Bloomfield, P. (1976). *Fourier Analysis of Time Series: An Introduction*. Wiley.
- [26] Bojarski, N. N. (1982). The k-space formulation of the scattering problem in the time domain. *The Journal of the Acoustical Society of America*, 72(2):570–584.
- [27] Bojarski, N. N. (1985). The k-space formulation of the scattering problem in the time domain: An improved single propagator formulation. *The Journal of the Acoustical Society of America*, 77(3):826–831.
- [28] Bressan, A. and Takacs, S. (2019). Sum factorization techniques in Isogeometric Analysis. *Computer Methods in Applied Mechanics and Engineering*, 352:437–460.
- [29] Bubrick, E. J., McDannold, N. J., Orozco, J., Mariano, T. Y., Rigolo, L., Golby, A. J., Tie, Y., and White, P. J. (2024). Transcranial ultrasound neuromodulation for epilepsy: A pilot safety trial. *Brain Stimulation*, 17(1).

- [30] Butcher, J. C. (2016). *Numerical methods for ordinary differential equations*. John Wiley & Sons.
- [31] Cambronero, S., Dupré, A., Mastier, C., and Melodelima, D. (2023). Non-invasive high-intensity focused ultrasound treatment of liver tissues in an in vivo porcine model: Fast, large and safe ablations using a toroidal transducer. *Ultrasound in Medicine Biology*, 49(1):212–224.
- [32] Chan, E., Heimlich, M., Purkayastha, A., and van de Geijn, R. (2007). Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783.
- [33] Chin-Joe-Kong, M. J. S., Mulder, W. A., and Van Veldhuizen, M. (1999). Higher-order triangular and tetrahedral finite elements with mass lumping for solving the wave equation. *Journal of Engineering Mathematics*, 35(4):405–426.
- [34] Choi, V., Rohn, J. L., Stoodley, P., Carugo, D., and Stride, E. (2023). Drug delivery strategies for antibiofilm therapy. *Nature Reviews Microbiology*, 21(9).
- [35] Ciarlet, P. G. (1978). *The finite element method for elliptic problems*. SIAM.
- [36] Clement, G. and Hynynen, K. (2003). Forward planar projection through layered media. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 50(12):1689–1698.
- [37] Clement, G. T. and Hynynen, K. (2000). Field characterization of therapeutic ultrasound phased arrays through forward and backward planar projection. *The Journal of the Acoustical Society of America*, 108(1):441–446.
- [38] Clement, G. T. and Hynynen, K. (2002). Correlation of ultrasound phase with physical skull properties. *Ultrasound in Medicine and Biology*, 28(5):617–624.
- [39] Cleveland, R. O., Hamilton, M. F., and Blackstock, D. T. (1996). Time-domain modeling of finite-amplitude sound in relaxing fluids. *The Journal of the Acoustical Society of America*, 99(6):3312–3318.
- [40] Cordeiro, E. R., Cathelineau, X., Thüroff, S., Marberger, M., Crouzet, S., and de la Rosette, J. J. (2012). High-intensity focused ultrasound (HIFU) for definitive treatment of prostate cancer. *BJU International*, 110(9):1228–1242.
- [41] Coussios, C. C. and Roy, R. A. (2008). Applications of Acoustics and Cavitation to Noninvasive Therapy and Drug Delivery. *Annual Review of Fluid Mechanics*, 40(1):395–420.
- [42] Cox, B. T. and Beard, P. C. (2005). Fast calculation of pulsed photoacoustic fields in fluids using k-space methods. *The Journal of the Acoustical Society of America*, 117(6):3616–3627.
- [43] Cueto, C., Bates, O., Strong, G., Cudeiro, J., Loporini, F., Agudo, O. C., Gorman, G., Guasch, L., and Tang, M.-X. (2022). Stride: a flexible platform for high-performance ultrasound computed tomography.
- [44] Curra, F., Mourad, P., Khokhlova, V., Cleveland, R., and Crum, L. (2000). Numerical simulations of heating patterns and tissue temperature response due to high-intensity focused ultrasound. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 47(4):1077–1089.
- [45] Dababou, S., Marrocchio, C., Rosenberg, J., Bitton, R., Pauly, K. B., Napoli, A., Hwang, J. H., and Ghanouni, P. (2017). A meta-analysis of palliative treatment of pancreatic cancer with high intensity focused ultrasound. *Journal of Therapeutic Ultrasound*, 5(1):9.

- [46] Dean, J. (2023). *Mathematical and computational aspects of solving mixed-domain problems using the finite element method*. PhD thesis, University of Cambridge.
- [47] Demi, L. (2013). *Modeling nonlinear propagation of ultrasound through inhomogeneous biomedical media*. PhD thesis, Delft University of Technology, Netherlands.
- [48] Demi, L., Dongen, K. W. A. v., and Verweij, M. D. (2011). A contrast source method for nonlinear acoustic wave fields in media with spatially inhomogeneous attenuation. *The Journal of the Acoustical Society of America*, 129(3):1221–1230.
- [49] Demi, L. and Verweij, M. (2014). Nonlinear Acoustics. *Comprehensive Biomedical Physics*, pages 387–399.
- [50] Doinikov, A. A., Novell, A., Calmon, P., and Bouakaz, A. (2014). Simulations and measurements of 3-D ultrasonic fields radiated by phased-array transducers using the Westervelt equation. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 61(9):1470–1477.
- [51] Duck, F. A. (1990). Chapter 4 - Acoustic Properties of Tissue at Ultrasonic Frequencies. In Duck, F. A., editor, *Physical Properties of Tissues*, pages 73–135. Academic Press, London.
- [52] Duczek, S. and Gravenkamp, H. (2019). Mass lumping techniques in the spectral element method: On the equivalence of the row-sum, nodal quadrature, and diagonal scaling methods. *Computer Methods in Applied Mechanics and Engineering*, 353:516–569.
- [53] Engquist, B. and Majda, A. (1977). Absorbing boundary conditions for numerical simulation of waves. *Proceedings of the National Academy of Sciences*, 74:1765–1766.
- [54] Farrell, P. E. (2021). Finite Element Methods for PDEs. <https://people.maths.ox.ac.uk/farrellp/femvideos/notes.pdf>.
- [55] Fichtner, A. (2010). *Full seismic waveform modelling and inversion*. Springer Science & Business Media.
- [56] Fonov, V., Evans, A., McKinstry, R., Almlı, C., and Collins, D. (2009). Unbiased nonlinear average age-appropriate brain templates from birth to adulthood. *NeuroImage*, 47(Supplement 1):S102.
- [57] Fonov, V., Evans, A. C., Botteron, K., Almlı, C. R., McKinstry, R. C., and Collins, D. L. (2011). Unbiased average age-appropriate atlases for pediatric studies. *NeuroImage*, 54(1):313–327.
- [58] Frazier, N., Payne, A., de Bever, J., Dillon, C., Panda, A., Subrahmanyam, N., and Ghandehari, H. (2016). High intensity focused ultrasound hyperthermia for enhanced macromolecular delivery. *Journal of Controlled Release*, 241:186–193.
- [59] Fry, F. J. and Barger, J. E. (1978). Acoustical properties of the human skull. *The Journal of the Acoustical Society of America*, 63(5):1576–1590.
- [60] Geevers, S., Mulder, W. A., and van der Vegt, J. J. W. (2018). New Higher-Order Mass-Lumped Tetrahedral Elements for Wave Propagation Modelling. *SIAM Journal on Scientific Computing*, 40(5):A2830–A2857.
- [61] Gelat, P. (2014). *Focusing the field of a HIFU array transducer through human ribs*. PhD thesis, UCL (University College London).
- [62] Geuzaine, C. and Remacle, J.-F. (2009). Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331.

- [63] Ginter, S., Liebler, M., Steiger, E., Dreyer, T., and Riedlinger, R. E. (2002). Full-wave modeling of therapeutic ultrasound: Nonlinear ultrasound propagation in ideal fluids. *The Journal of the Acoustical Society of America*, 111(5):2049–2059.
- [64] Goodman, J. W. (2005). *Introduction to Fourier Optics*. Roberts and Company Publishers.
- [65] Grote, M. J., Mehlin, M., and Mitkova, T. (2015). Runge–Kutta-based explicit local time-stepping methods for wave propagation. *SIAM Journal on Scientific Computing*, 37:A747–a775.
- [66] Groth, S. (2022). Volume INtegral Equation Solver (VINES).
- [67] Groth, S. P., G  lat, P., Haqshenas, S. R., Saffari, N., van ’t Wout, E., Betcke, T., and Wells, G. N. (2021). Accelerating frequency-domain numerical methods for weakly nonlinear focused ultrasound using nested meshes. *The Journal of the Acoustical Society of America*, 150(1):441–453.
- [68] Gu, J. and Jing, Y. (2015). Modeling of wave propagation for medical ultrasound: A review. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 62:1979–1993.
- [69] G  lat, P., Haqshenas, R., and van ’t Wout, E. (2022). OptimUS.
- [70] Haddadi, S. and Ahmadian, M. T. (2018). Analysis of nonlinear acoustic wave propagation in HIFU treatment using Westervelt equation. *Scientia Iranica*, 25:2087–2097.
- [71] H  ggblad, J. and Runborg, O. (2014). Accuracy of staircase approximations in finite-difference methods for wave propagation. *Numerische Mathematik*, 128(4):741–771.
- [72] Hairer, E. and N  rsett, S. P. and Wanner, G. (1993). *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, Berlin, Heidelberg.
- [73] Hallaj, I. M. and Cleveland, R. O. (1999). FDTD simulation of finite-amplitude pressure and temperature fields for biomedical ultrasound. *The Journal of the Acoustical Society of America*, 105(5):L7–L12.
- [74] Hamilton, M. and Blackstock, D. (1998). *Nonlinear Acoustics*. Academic Press.
- [75] Haqshenas, S., G  lat, P., van ’t Wout, E., Betcke, T., and Saffari, N. (2021). A fast full-wave solver for calculating ultrasound propagation in the body. *Ultrasonics*, 110:106240.
- [76] Hettinga, J. K., Lyons, B., Balkaran, J., Rijal, P., Dunn-Lawless, D., Caproni, L., Gray, M., Suslick, K. S., Coussios, C. C., and Carlisle, R. C. (2023). Cavitation-mediated transcutaneous delivery of protein and nucleotide-based antigen for rapid high-level immune responses. *Advanced Therapeutics*, 6(12).
- [77] Hodges, R. (2011). *Underwater Acoustics: Analysis, Design and Performance of Sonar*. Wiley.
- [78] Hoefler, T. and Traff, J. L. (2009). Sparse collective operations for MPI. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8.
- [79] Hoffelner, J., Landes, H., Kaltenbacher, M., and Lerch, R. (2001). Finite element simulation of nonlinear wave propagation in thermoviscous fluids including dissipation. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 48(3):779–786.
- [80] Huijssen, J. (2008). *Modeling of Nonlinear Medical Diagnostic Ultrasound*. Phd thesis, Delft University of Technology.

- [81] Huijssen, J. and Verweij, M. D. (2010). An iterative method for the computation of nonlinear, wide-angle, pulsed acoustic fields of medical diagnostic transducers. *The Journal of the Acoustical Society of America*, 127(1):33–44.
- [82] Humphrey, V. F. (2000). Nonlinear propagation in ultrasonic fields: measurements, modelling and harmonic imaging. *Ultrasonics*, 38(1):267–272.
- [83] Hynynen, K., McDannold, N., Vykhodtseva, N., and Jolesz, F. A. (2001). Noninvasive MR imaging-guided focal opening of the blood-brain barrier in rabbits. *Radiology*, 220(3):640–646.
- [84] Ihlenburg, F. (1998). *Finite element analysis of acoustic scattering*. Springer.
- [85] Ihlenburg, F. and Babuška, I. (1995). Dispersion analysis and error estimation of Galerkin finite element methods for the Helmholtz equation. *International Journal for Numerical Methods in Engineering*, 38(22):3745–3774.
- [86] Jaros, J., Rendell, A. P., and Treeby, B. E. (2016). Full-wave nonlinear ultrasound simulation on distributed clusters with applications in high-intensity focused ultrasound. *The International Journal of High Performance Computing Applications*, 30(2):137–155.
- [87] Jenne, J. W., Preusser, T., and Günther, M. (2012). High-intensity focused ultrasound: Principles, therapy guidance, simulations and applications. *Zeitschrift für Medizinische Physik*, 22(4):311–322. Schwerpunkt: Multimodale Bildgebung und Therapie.
- [88] Jindal, S., Jung, J., Lee, K., and Chern, B. (2023). High-intensity focused ultrasound for the treatment of fibroids: A single-center experience in singapore. *Gynecol Minim Invasive Ther*, 12(1):15–25.
- [89] Jing, Y., Tao, M., and Cannata, J. (2014). An improved wave-vector frequency-domain method for nonlinear wave modeling. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 61(3):515–524.
- [90] Jing, Y., Tao, M., and Clement, G. T. (2011). Evaluation of a wave-vector-frequency-domain method for nonlinear wave propagation. *The Journal of the Acoustical Society of America*, 129(1):32–46.
- [91] Johns, L. D. (2002). Nonthermal effects of therapeutic ultrasound: the frequency resonance hypothesis. *J. Athl. Train.*, 37(3):293–299.
- [92] Johnson, S. G. (2021). Notes on Perfectly Matched Layers (PMLs).
- [93] Kaltenbacher, B. and Shevchenko, I. (2019). Well-posedness of the Westervelt equation with higher order absorbing boundary conditions. *Journal of Mathematical Analysis and Applications*, 479:1595–1617.
- [94] Karniadakis, G. and Sherwin, S. (2005). *Spectral/hp Element Methods for Computational Fluid Dynamics: Second Edition*. Numerical Mathematics and Scie. OUP Oxford.
- [95] Kashcheeva, S. S., Sapozhnikov, O. A., Khokhlova, V. A., Averkiou, M. A., and Crum, L. A. (2000). Nonlinear distortion and attenuation of intense acoustic waves in lossy media obeying a frequency power law. *Acoustical Physics*, 46:170–177.
- [96] Kennedy, C. A., Carpenter, M. H., and Lewis, R. (2000). Low-storage, explicit Runge—Kutta schemes for the compressible Navier—Stokes equations. *Applied Numerical Mathematics*, 35(3):177–219.

- [97] Kennedy, J. E., ter Haar, G. R., and Cranston, D. (2003). High intensity focused ultrasound: surgery of the future? *The British Journal of Radiology*, 76(909):590–599.
- [98] Ketcheson, D. I. (2010). Runge–Kutta methods with minimum storage implementations. *Journal of Computational Physics*, 229(5):1763–1773.
- [99] Khokhlova, V. A., Souchon, R., Tavakkoli, J., Sapozhnikov, O. A., and Cathignol, D. (2001). Numerical modeling of finite-amplitude sound beams: Shock formation in the near field of a CW plane piston source. *The Journal of the Acoustical Society of America*, 110(1):95–108.
- [100] Kolev, T., Fischer, P., Min, M., Dongarra, J., Brown, J., Dobrev, V., Warburton, T., Tomov, S., Shephard, M. S., Abdelfattah, A., Barra, V., Beams, N., Camier, J.-S., Chalmers, N., Dudouit, Y., Karakus, A., Karlin, I., Kerkemeier, S., Lan, Y.-H., Medina, D., Merzari, E., Obabko, A., Pazner, W., Rathnayake, T., Smith, C. W., Spies, L., Swirydowicz, K., Thompson, J., Tomboulides, A., and Tomov, V. (2021). Efficient exascale discretizations: High-order finite element methods. *The International Journal of High Performance Computing Applications*, 35(6):527–552.
- [101] Komatitsch, D. and Tromp, J. (1999). Introduction to the spectral element method for three-dimensional seismic wave propagation. *Geophysical Journal International*, 139(3):806–822.
- [102] Komatitsch, D., Tsuboi, S., Tromp, J., Levander, A., and Nolet, G. (2005). The spectral-element method in seismology. *Geophysical Monograph-American Geophysical Union*, 157:205.
- [103] Kothapalli, S. V. V. N., Partanen, A., Zhu, L., Altman, M. B., Gach, H. M., Hallahan, D. E., and Chen, H. (2018). A convenient, reliable, and fast acoustic pressure field measurement method for magnetic resonance-guided high-intensity focused ultrasound systems with phased array transducers. *Journal of Therapeutic Ultrasound*, 6(1):5.
- [104] Kowalczyk, K. and van Walstijn, M. (2011). Room Acoustics Simulation Using 3-D Compact Explicit FDTD Schemes. *IEEE Transactions on Audio, Speech, and Language Processing*, 19(1):34–46.
- [105] Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: a llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM ’15. Association for Computing Machinery.
- [106] Langtangen, H. P. and Mardal, K.-A. (2019). *Introduction to Numerical Methods for Variational Problems*, volume 21. Springer Nature.
- [107] Lee, Y. and Hamilton, M. F. (1995). Time-domain modeling of pulsed finite-amplitude sound beams. *The Journal of the Acoustical Society of America*, 97(2):906–917.
- [108] Lemprière, S. (2023). Ultrasound lets gene therapy into the brain. *Nature Reviews Neurology*, 19(6):326–326.
- [109] Leung, S. A., Moore, D., Webb, T. D., Snell, J., Ghanouni, P., and Butts Pauly, K. (2021). Transcranial focused ultrasound phase correction using the hybrid angular spectrum method. *Scientific Reports*, 11(1):6532.
- [110] Maier, M. and Kronbichler, M. (2021). Efficient Parallel 3D Computation of the Compressible Euler Equations with an Invariant-Domain Preserving Second-Order Finite-Element Scheme. *ACM Trans. Parallel Comput.*, 8(3).
- [111] Marty, P., Boehm, C., Paverd, C., Rominger, M., and Fichtner, A. (2022). Full-waveform ultrasound modeling of soft tissue-bone interactions using conforming hexahedral meshes. In Zhao, W. and Yu, L., editors, *Medical Imaging 2022: Physics of Medical Imaging*, volume 12031, page 120313H. International Society for Optics and Photonics, SPIE.

- [112] Mast, T. D. (2000). Empirical relationships between acoustic parameters in human soft tissues. *Acoustics Research Letters Online*, 1(2):37–42.
- [113] McMahon, D., O'Reilly, M. A., and Hynynen, K. (2021). Therapeutic Agent Delivery Across the Blood–Brain Barrier Using Focused Ultrasound. *Annual Review of Biomedical Engineering*, 23(1):89–113.
- [114] Melenk, J., Gerdes, K., and Schwab, C. (2001). Fully discrete hp-finite elements: fast quadrature. *Computer Methods in Applied Mechanics and Engineering*, 190(32):4339–4364.
- [115] Mendousse, J. S. (1953). Nonlinear Dissipative Distortion of Progressive Sound Waves at Moderate Amplitudes. *The Journal of the Acoustical Society of America*, 25(1):51–54.
- [116] Meng, Y., Hynynen, K., and Lipsman, N. (2021). Applications of focused ultrasound in the brain: from thermoablation to drug delivery. *Nature Reviews Neurology*, 17(1):7–22.
- [117] Michael Gray, L. S. and Coussios, C. (2022). Effects of human tissue acoustic properties, abdominal wall shape, and respiratory motion on ultrasound-mediated hyperthermia for targeted drug delivery to pancreatic tumors. *International Journal of Hyperthermia*, 39(1):918–934.
- [118] Niu, Y., Wang, Y., Meng, H., Yin, C., Dang, K., and Qian, A. (2024). Chapter 11 - Biomechanics in clinical application for bone diseases. In Qian, A. and Hu, L., editors, *Bone Cell Biomechanics, Mechanobiology and Bone Diseases*, pages 315–352. Academic Press.
- [119] Orszag, S. A. (1980). Spectral methods for problems in complex geometries. *Journal of Computational Physics*, 37(1):70–92.
- [120] Oskooi, A. F., Avniel, Y., Zhang, L., and Johnson, S. G. (2008). The failure of perfectly matched layers, and towards their redemption by adiabatic absorbers. *Optics Express, Vol. 16, Issue 15*, pp. 11376–11392, 16:11376–11392.
- [121] P Gélât and G ter Haar and N Saffari (2012). Scattering of the field of a multi-element phased array by human ribs. *Journal of Physics: Conference Series*, 353:012010.
- [122] Park, H., Kim, E., Kim, J., Ro, Y., and Ko, J. (2015). High-Intensity Focused Ultrasound for the Treatment of Wrinkles and Skin Laxity in Seven Different Facial Areas. *Ann Dermatol*, 27(6):688–693.
- [123] Pennes, H. H. (1948). Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm. *Journal of Applied Physiology*, 1(2):93–122.
- [124] Pichardo, S., Sin, V. W., and Hynynen, K. (2010). Multi-frequency characterization of the speed of sound and attenuation coefficient for longitudinal transmission of freshly excised human skulls. *Physics in Medicine & Biology*, 56(1):219.
- [125] Pietroni, N., Campen, M., Sheffer, A., Cherchi, G., Bommers, D., Gao, X., Scateni, R., Ledoux, F., Remacle, J., and Livesu, M. (2022). Hex-Mesh Generation and Processing: A Survey. *ACM Trans. Graph.*, 42(2).
- [126] Pind, F., Engsig-Karup, A. P., Jeong, C.-H., Hesthaven, J. S., Mejling, M. S., and Strømman-Andersen, J. (2019). Time domain room acoustic simulations using the spectral element method. *The Journal of the Acoustical Society of America*, 145(6):3299–3310.
- [127] Pinton, G. (2021). A fullwave model of the nonlinear wave equation with multiple relaxations and relaxing perfectly matched layers for high-order numerical finite-difference solutions.

- [128] Pinton, G., Aubry, J.-F., Bossy, E., Muller, M., Pernot, M., and Tanter, M. (2012). Attenuation, scattering, and absorption of ultrasound in the skull bone. *Medical Physics*, 39(1):299–307.
- [129] Pinton, G. F., Dahl, J., Rosenzweig, S., and Trahey, G. E. (2009). A heterogeneous nonlinear attenuating full-wave model of ultrasound. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 56(3):474–488.
- [130] Pinton, G. F. and Trahey, G. E. (2008). Modeling of Shock Wave Propagation in Large Amplitude Ultrasound. *Ultrasonic Imaging*, 30(1):44–60. PMID: 18564596.
- [131] Poya, R., Gil, A. J., and Ortigosa, R. (2017). A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics. *Computer Physics Communications*.
- [132] Pulkkinen, A., Werner, B., Martin, E., and Hynynen, K. (2014). Numerical simulations of clinical focused ultrasound functional neurosurgery. *Physics in Medicine Biology*, 59(7):1679.
- [133] Qiao, S., Jackson, E., Coussios, C. C., and Cleveland, R. O. (2016). Simulation of nonlinear propagation of biomedical ultrasound using pzfex and the Khokhlov-Zabolotskaya-Kuznetsov Texas code. *The Journal of the Acoustical Society of America*, 140(3):2039–2046.
- [134] Ran, L.-F., Xie, X.-P., Xia, J.-Z., Xie, F.-L., Fan, Y.-M., and Wu, F. (2023). T-lymphocytes from focused ultrasound ablation subsequently mediate cellular antitumor immunity after adoptive cell transfer immunotherapy. *Frontiers in Immunology*, 14.
- [135] Ries, M., de Senneville, B. D., Roujol, S., Berber, Y., Quesson, B., and Moonen, C. (2010). Real-time 3D target tracking in MRI guided focused ultrasound ablations in moving tissues. *Magnetic Resonance in Medicine*, 64(6):1704–1712.
- [136] Robertson, J. L. B., Cox, B. T., Jaros, J., and Treeby, B. E. (2017). Accurate simulation of transcranial ultrasound propagation for ultrasonic neuromodulation and stimulation. *The Journal of the Acoustical Society of America*, 141(3):1726–1738.
- [137] Roehl, T., Treibig, J., Hager, G., and Wellein, G. (2014). Overhead Analysis of Performance Counter Measurements. In *43rd International Conference on Parallel Processing Workshops (ICCPW)*, pages 176–185.
- [138] Schöberl, J. (2014). C++11 Implementation of Finite Elements in NGSolve.
- [139] Schot, S. H. (1992). Eighty years of Sommerfeld’s radiation condition. *Historia Mathematica*, 19:385–401.
- [140] Scroggs, M. W., Baratta, I. A., Richardson, C. N., and Wells, G. N. (2022a). Basix: a runtime finite element basis evaluation library. *Journal of Open Source Software*, 7(73):3982.
- [141] Scroggs, M. W., Dokken, J. S., Richardson, C. N., and Wells, G. N. (2022b). Construction of Arbitrary Order Finite Element Degree-of-Freedom Maps on Polygonal and Polyhedral Cell Meshes. *ACM Trans. Math. Softw.*, 48(2).
- [142] Sherwin, S. and Karniadakis, G. (1995). A triangular spectral element method; applications to the incompressible Navier–Stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 123(1):189–229.
- [143] Shevchenko, I. and Kaltenbacher, B. (2015). Absorbing boundary conditions for nonlinear acoustics: The Westervelt equation. *Journal of Computational Physics*, 302:200–221.
- [144] Smith, J. O. (accessed <date>). *Spectral Audio Signal Processing*. <http://ccrma.stanford.edu/~jos/sasp/>. online book, 2011 edition.

- [145] Solovchuk, M., Sheu, T. W. H., and Thiriet, M. (2013). Simulation of nonlinear Westervelt equation for the investigation of acoustic streaming and nonlinear propagation effects. *The Journal of the Acoustical Society of America*, 134:3931.
- [146] Sommerfeld, A. (1912). Die Greensche Funktion der Schwingungsgleichung. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 21:309–352.
- [147] Soneson, J. (2019). HITU_Simulator.
- [148] Soneson, J. E. (2017). Extending the Utility of the Parabolic Approximation in Medical Ultrasound Using Wide-Angle Diffraction Modeling. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 64(4):679–687.
- [149] Sticko, S. and Kreiss, G. (2019). Higher order cut finite elements for the wave equation. *Journal of Scientific Computing*, 80(3):1867–1887.
- [150] Suomi, V., Jaros, J., Treeby, B., and Cleveland, R. (2016). Nonlinear 3-D simulation of high-intensity focused ultrasound therapy in the Kidney. *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBS*, 2016-October:5648–5651.
- [151] Tabei, M., Mast, T. D., and Waag, R. C. (2002). A k-space method for coupled first-order acoustic propagation equations. *The Journal of the Acoustical Society of America*, 111(1):53–63.
- [152] Tambova, A., Groth, S. P., White, J. K., and Polimeridis, A. G. (2018). Adiabatic Absorbers in Photonics Simulations with the Volume Integral Equation Method. *Journal of Lightwave Technology*, 36:3765–3777.
- [153] Tavakkoli, J., Cathignol, D., Souchon, R., and Sapozhnikov, O. A. (1998). Modeling of pulsed finite-amplitude focused sound beams in time domain. *The Journal of the Acoustical Society of America*, 104(4):2061–2072.
- [154] ter Haar, G. and Coussios, C. (2007). High intensity focused ultrasound: Physical principles and devices. *International Journal of Hyperthermia*, 23(2):89–104.
- [155] Tjøtta, J. N., Tjøtta, S., and Vefring, E. H. (1991). Effects of focusing on the nonlinear interaction between two collinear finite amplitude sound beams. *The Journal of the Acoustical Society of America*, 89(3):1017–1027.
- [156] Tomasian, A., Cazzato, R. L., Sharma, K., Gangi, A., and Jennings, J. W. (2023). Benign bone tumors: State of the art in minimally invasive percutaneous interventions. *RadioGraphics*, 43(2).
- [157] Tornberg, A.-K. and Engquist, B. (2008). Consistent boundary conditions for the Yee scheme. *Journal of Computational Physics*, 227(14):6922–6943.
- [158] Treeby, B. (2022). Benchmark problems for transcranial ultrasound simulation: Intercomparison library.
- [159] Treeby, B., Cox, B., and Jaros, J. (2022). Alternative acoustics software. <http://www.k-wave.org/acousticsoftware.php>. Accessed: 2023-06-21.
- [160] Treeby, B. E. and Cox, B. T. (2010a). k-Wave: MATLAB toolbox for the simulation and reconstruction of photoacoustic wave fields. *Journal of Biomedical Optics*, 15(2):021314.
- [161] Treeby, B. E. and Cox, B. T. (2010b). Modeling power law absorption and dispersion for acoustic propagation using the fractional Laplacian. *The Journal of the Acoustical Society of America*, 127(5):2741–2748.

- [162] Treeby, B. E. and Cox, B. T. (2014). Modeling power law absorption and dispersion in viscoelastic solids using a split-field and the fractional Laplaciana). *The Journal of the Acoustical Society of America*, 136:1499.
- [163] Treeby, B. E., Jaros, J., Rendell, A. P., and Cox, B. T. (2012). 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*, 131:4324.
- [164] Treibig, J., Hager, G., and Wellein, G. (2010). LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA.
- [165] Treibig, J., Hager, G., and Wellein, G. (2012). likwid-bench: An Extensible Microbenchmarking Platform for x86 Multicore Compute Nodes. In Brunst, H., Müller, M. S., Nagel, W. E., and Resch, M. M., editors, *Tools for High Performance Computing 2011*, pages 27–36, Berlin, Heidelberg. Springer Berlin Heidelberg.
- [166] Tsuchiya, T. and Kagawa, Y. (1991). Finite Element Analysis of Focusing Field in Nonlinear Acoustic Waves. *Japanese Journal of Applied Physics*, 30(S1):51.
- [167] van 't Wout, E., G  lat, P., Betcke, T., and Arridge, S. (2015). A fast boundary element method for the scattering analysis of high-intensity focused ultrasound. *The Journal of the Acoustical Society of America*, 138(5):2726–2737.
- [168] van 't Wout, E., Haqshenas, R., G  lat, P., and Saffari, N. (2023). Fast and accurate boundary element methods for large-scale computational acoustics. *The Journal of the Acoustical Society of America*, 154(4_supplement):A179–A179.
- [169] van 't Wout, E., Haqshenas, S. R., G  lat, P., Betcke, T., and Saffari, N. (2021). Benchmarking preconditioned boundary integral formulations for acoustics. *International Journal for Numerical Methods in Engineering*, 122(20):5873–5897.
- [170] van 't Wout, E., Haqshenas, S. R., G  lat, P., Betcke, T., and Saffari, N. (2022). Boundary integral formulations for acoustic modelling of high-contrast media. *Computers & Mathematics with Applications*, 105:136–149.
- [171] van 't Wout, E., Haqshenas, S. R., G  lat, P., Betcke, T., and Saffari, N. (2022). Frequency-robust preconditioning of boundary integral equations for acoustic transmission. *Journal of Computational Physics*, 462:111229.
- [172] Varslot, T. and Taraldsen, G. (2005). Computer simulation of forward wave propagation in soft tissue. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 52(9):1473–1482.
- [173] Verweij, M., Treeby, B., van Dongen, K., and Demi, L. (2014). Simulation of Ultrasound Fields. *Comprehensive Biomedical Physics*, pages 465–500.
- [174] Vyas, U. and Christensen, D. (2012). Ultrasound beam simulations in inhomogeneous tissue geometries using the hybrid angular spectrum method. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 59(6):1093–1100.
- [175] Vyas, U. and Christensen, D. A. (2011). Extension of the angular spectrum method to calculate pressure from a spherically curved acoustic source. *The Journal of the Acoustical Society of America*, 130(5):2687–2693.

- [176] Walsh, T. and Torres, M. (2007). Finite element methods for nonlinear acoustics in fluids. *Journal of Computational Acoustics*, 15(03):353–375.
- [177] Wanner, G. and Hairer, E. (1996). *Solving Ordinary Differential Equations II*, volume 375. Springer Berlin Heidelberg New York.
- [178] Webb, T. D., Leung, S. A., Ghanouni, P., Dahl, J. J., Pelc, N. J., and Pauly, K. B. (2021). Acoustic Attenuation: Multifrequency Measurement and Relationship to CT and MR Imaging. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 68(5):1532–1545.
- [179] White, E., Myhre, S., Serafini, M. R., Broad, M., Chesnut, A., Browning, M., Heishman, D., Knupp, J., and Chao, J. C. (2024). State of the Field Report 2024.
- [180] White, P. J., Clement, G. T., and Hynynen, K. (2006). Longitudinal and shear mode ultrasound propagation in human skull bone. *Ultrasound in Medicine and Biology*, 32(7):1085–1096.
- [181] Williamson, J. (1980). Low-storage Runge–Kutta schemes. *Journal of Computational Physics*, 35(1):48–56.
- [182] Wise, E. S., Robertson, J. L. B., Cox, B. T., and Treeby, B. E. (2017). Staircase-free acoustic sources for grid-based models of wave propagation. In *2017 IEEE International Ultrasonics Symposium (IUS)*, pages 1–4.
- [183] Wu, F., Wang, Z.-B., Chen, W.-Z., Zhu, H., Bai, J., Zou, J.-Z., Li, K.-Q., Jin, C.-B., Xie, F.-L., and Su, H.-B. (2004). Extracorporeal High Intensity Focused Ultrasound Ablation in the Treatment of Patients with Large Hepatocellular Carcinoma. *Annals of Surgical Oncology*, 11(12):1061–1069.
- [184] Wu, P. and Stepinski, T. (1999). Extension of the angular spectrum approach to curved radiators. *The Journal of the Acoustical Society of America*, 105(5):2618–2627.
- [185] Wu, Y., Liao, S., and Zhao, X. (2005). Some notes on the general boundary element method for highly nonlinear problems. *Communications in Nonlinear Science and Numerical Simulation*, 10(7):725–735.
- [186] Wunker, C., Piorkowska, K., Keunen, B., Babichev, Y., Wong, S. M., Regenold, M., Dunne, M., Nomikos, J., Siddiqui, M., Pichardo, S., Foltz, W., Waspe, A. C., Gerstle, J. T., Drake, J. M., and Gladdy, R. A. (2023). Magnetic resonance-guided high intensity focused ultrasound generated hyperthermia: A feasible treatment method in a murine rhabdomyosarcoma model. *JoVE*, (191):e64544.
- [187] Younan, Y., Deffieux, T., Larrat, B., Fink, M., Tanter, M., and Aubry, J.-F. (2013). Influence of the pressure field distribution in transcranial ultrasonic neurostimulation. *Medical Physics*, 40(8):082902.
- [188] Young, L. C. (2019). Orthogonal collocation revisited. *Computer Methods in Applied Mechanics and Engineering*, 345:1033–1076.
- [189] Yuldashev, P. V., Karzova, M. M., Kreider, W., Rosnitskiy, P. B., Sapozhnikov, O. A., and Khokhlova, V. A. (2021). “HIFU Beam:” A Simulator for Predicting Axially Symmetric Nonlinear Acoustic Fields Generated by Focused Transducers in a Layered Medium. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 68(9):2837–2852.
- [190] Yuldashev, P. V. and Khokhlova, V. A. (2011). Simulation of three-dimensional nonlinear fields of ultrasound therapeutic arrays. *Acoustical Physics*, 57(3):334–343.

- [191] Yuldashev, P. V., Mezdokhin, I. S., and Khokhlova, V. A. (2018). Wide-Angle Parabolic Approximation for Modeling High-Intensity Fields from Strongly Focused Ultrasound Transducers. *Acoustical Physics*, 64:309–319.
- [192] Zhen Xu, Timothy L. Hall, E. V. and Jr., F. T. L. (2021). Histotripsy: the first noninvasive, non-ionizing, non-thermal ablation technique based on ultrasound. *International Journal of Hyperthermia*, 38(1):561–575.
- [193] Zhou, Y.-F. (2011). High intensity focused ultrasound in clinical tumor ablation. *World J Clin Oncol*, 2(1):8–27.
- [194] Zhu, L., Altman, M. B., Laszlo, A., Straube, W., Zoberi, I., Hallahan, D. E., and Chen, H. (2019). Ultrasound hyperthermia technology for radiosensitization. *Ultrasound in Medicine Biology*, 45(5):1025–1043.

Appendix A

Packing and unpacking CUDA kernels

In section 7.4, the implementation to utilise a multi-GPU system was presented. In particular, the scatter forward and scatter reverse implementation that enable the utilisation of multiple NVIDIA GPUs. This section presents the details of the packing and unpacking kernels of the scatter forward and scatter reverse implementation.

A.1 Scatter forward

A.1.1 Packing

The packing kernel extracts the DOF from the buffer (local DOF) to be stored before being send to other processes. Listing A.1 shows the Numba CUDA packing implementation for scatter forward.

```
1 @cuda.jit
2 def pack_fwd(
3     in_: numba.types.Array,
4     out_: numba.types.Array,
5     index: numba.types.Array
6 ):
7
8     """
9     Pack coefficient.
10
11     Parameters
12     -----
13     in_ : input array
14     out_ : output array
15     index : indices of the input array to pack
16     """
17
18     thread_id = cuda.threadIdx.x
19     block_id = cuda.blockIdx.x
20     idx = thread_id + block_id * cuda.blockDim.x
```

```

21
22     if idx < index.size:
23         out_[idx] = in_[index[idx]]

```

Listing A.1 Packing kernel for scatter forward function.

A.1.2 Unpacking

The unpacking kernel for scatter forward replaces the DOF in the buffer (local DOF) using the data receive from other processes. Listing A.2 shows the Numba CUDA unpacking implementation for scatter forward.

```

1  @cuda.jit
2  def unpack_fwd(
3      in_: numba.types.Array,
4      out_: numba.types.Array,
5      index: numba.types.Array, N: int
6  ):
7      """
8      Unpack coefficient.
9
10     Parameters
11     -----
12     in_ : input array
13     out_ : output array
14     index : indices of the output array to unpack
15     """
16
17     thread_id = cuda.threadIdx.x
18     block_id = cuda.blockIdx.x
19     idx = thread_id + block_id * cuda.blockDim.x
20
21     if idx < index.size:
22         out_[index[idx] + N] = in_[idx]

```

Listing A.2 Unpacking kernel for scatter forward function.

A.2 Scatter reverse

A.2.1 Packing

The packing kernel of scatter reverse extracts the *ghost* DOF from the buffer (local DOF) to be stored before being send to other processes. Listing A.3 shows the Numba CUDA packing implementation for scatter reverse.

```

1  def pack_rev(
2      in_: numba.types.Array,

```

```

3     out_: numba.types.Array,
4     index: numba.types.Array,
5     N: int
6 ):
7     """
8     Pack coefficient.
9
10    Parameters
11    -----
12    in_ : input array
13    out_ : output array
14    index : indices of the input array to pack
15    """
16
17    thread_id = cuda.threadIdx.x
18    block_id = cuda.blockIdx.x
19    idx = thread_id + block_id * cuda.blockDim.x
20
21    if idx < index.size:
22        out_[idx] = in_[index[idx] + N]

```

Listing A.3 Packing kernel for scatter reverse function.

A.2.2 Unpacking

The unpacking kernel of scatter reverse adds the contribution of the *ghost* DOF from other processes into the buffer (local DOF). Listing A.4 shows the Numba CUDA unpacking implementation for scatter reverse.

```

1 @cuda.jit
2 def unpack_rev(
3     in_: numba.types.Array,
4     out_: numba.types.Array,
5     index: numba.types.Array
6 ):
7     """
8     Unpack coefficient.
9
10    Parameters
11    -----
12    in_ : input array
13    out_ : output array
14    index : indices of the output array to unpack
15    """
16
17    thread_id = cuda.threadIdx.x
18    block_id = cuda.blockIdx.x
19    idx = thread_id + block_id * cuda.blockDim.x

```

```
20
21     if idx < index.size:
22         cuda.atomic.add(out_, index[idx], in_[idx])
```

Listing A.4 Unpacking kernel for scatter reverse function.