ORBITAL RESONANCES AND GPU ACCELERATION OF BINARY BLACK HOLE INSPIRAL SIMULATIONS

by

Adam Gabriel Marcel Lewis

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Department of Physics
University of Toronto

© Copyright 2018 by Adam Gabriel Marcel Lewis
Abstract

Orbital resonances and GPU acceleration of binary black hole inspiral simulations

Adam Gabriel Marcel Lewis
Doctor of Philosophy
Department of Physics
University of Toronto
2018

Numerical relativity, the direct numerical integration of the Einstein field equations, is now a mature subfield of computational physics, playing a critical role in the generation of signal templates for comparison with data from ground-based gravitational wave detectors. The application of numerical relativity techniques to new problems is at present complicated by long wallclock times and intricate code. In this thesis we lay groundwork to improve this situation by presenting a GPU port of the numerical relativity code SpEC. Our port keeps code maintenance feasible by relying on various layers of automation, and achieves high performance across a variety of GPUs. We secondly introduce a C++ software package, TLoops, which allows numerical manipulation of tensors using single-line C++ source-code expressions resembling familiar tensor calculus notation. These expressions may be compiled and executed immediately, but also can be used to automatically generate equivalent GPU or low-level CPU code, which then executes in their place. The GPU code in particular achieves near-peak performance. Finally, we present simulations of eccentric binary black holes. We develop new methods to extract the fundamental frequencies of these systems. Using these frequencies we identify when these binaries pass through coordinate resonances, at which points high mass-ratio inspirals can experience short-timescale phase-dependent deviations from smooth inspiral called “kicks”. We find no evidence for such kicks at comparable mass-ratio.
For my mother, my late father, and my sister.
Acknowledgements

It is impossible to adequately thank the numerous colleagues, confidantes, and collaborators who made this work possible. First and foremost, I am indebted to my supervisor Harald Pfeiffer for his unwavering support. He has provided me with personal and professional skills I will use throughout my career. I am also thankful for the warm mentorship of Charles Dyer, along with the support of my collaborator Aaron Zimmerman. Finally, I would like to thank my mother Gail Coulas, my father David Lewis, and my sister Anne Lewis, for supporting their relative despite his odd career choices.
# Table of Contents

Dedication iii  
Acknowledgements iv  
Table of Contents v  

1 Overview 1  

2 Numerical Relativity Orientation 6  
   2.1 Introduction .............................................. 6  
   2.2 3+1 Formulation ........................................... 7  
       2.2.1 From an Initial Value Formulation to a Numerical Relativity Code 10  
   2.3 Applications of NR ......................................... 15  

3 GPU-Accelerated Simulations of Isolated Black Holes 18  
   3.1 Introduction .................................................. 18  
   3.2 Technological Overview ...................................... 23  
       3.2.1 The Spectral Einstein Code ................................. 23  
       3.2.2 Programming for NVIDIA GPUs ............................ 24  
       3.2.3 GPU Benchmarking ....................................... 27  
   3.3 Details of our port ........................................... 32  
       3.3.1 Overview .................................................. 32  
       3.3.2 Memory Management ....................................... 33  
       3.3.3 Jacobian multiplication .................................. 35  
       3.3.4 Spectral Operations: Differentiation and Filtering .......... 41  
       3.3.5 DataMesh Operations, Apply BCs, GHEqns .................. 54  
   3.4 Benchmarks of Overall Code ................................... 57  
   3.5 Conclusions ................................................... 58  
   3.6 Acknowledgments ............................................... 61
4 Tensor Loops

4.1 Introduction .................................................. 62
4.2 Direct evaluation of tensor loops using expression templates ........... 65
  4.2.1 Spectral Einstein Code ..................................... 65
  4.2.2 Tensors and SpEC’s Tensor class ............................ 66
  4.2.3 Capturing a TLoops-expression as a type ...................... 68
  4.2.4 Evaluation of TLoops-template .............................. 76
  4.2.5 Sum-operations ........................................... 79
4.3 Automatic code generation .................................... 81
4.4 Design Considerations of Automatically Generated CUDA Code ........... 86
4.5 Benchmarks .................................................... 91
  4.5.1 Methodology ............................................... 91
  4.5.2 Benchmarks of simple expressions ........................... 94
  4.5.3 Benchmarks of more complex expressions ...................... 97
  4.5.4 Impact of CPU compiler ................................... 100
  4.5.5 Impact of templating over Tensor-indices ..................... 100
4.6 Conclusion .................................................... 103

5 Frequencies and Resonances of Generic BBH Inspirals ...................... 104

5.1 Abstract ..................................................... 104
5.2 Introduction .................................................. 104
5.3 Motion in Kerr spacetimes .................................... 108
  5.3.1 Geodesic orbits in Kerr .................................... 108
  5.3.2 Fundamental frequencies of Kerr ............................ 110
  5.3.3 Orbital resonances ...................................... 111
5.4 Numerical relativity simulations ................................ 113
  5.4.1 Simulations of eccentric, precessing black hole binaries .......... 113
  5.4.2 Dynamics of simulated binaries ............................ 115
  5.4.3 Plots/Eccentricity decay ................................ 121
5.5 Equatorial binaries ........................................... 121
  5.5.1 Frequencies for equatorial orbits ........................... 122
  5.5.2 Results ................................................ 123
5.6 Inclined binaries .............................................. 125
  5.6.1 Definition of frequencies ................................ 125
  5.6.2 Precession rates ....................................... 128
  5.6.3 Resonances ............................................ 131
5.7 Conclusions ................................................. 136
5.8 Appendices ................................................. 138
  5.8.1 The Kerr metric ..................................... 138
  5.8.2 Envelope subtraction method in the slow time approximation .. 139
  5.8.3 Evolution of eccentricity and semi-major axis .............. 141
  5.8.4 Validation of frequency extraction methods .............. 142
  5.8.5 Estimation of resonant accretion ....................... 146

6 Conclusions ................................................. 148

Bibliography .................................................. 151
Chapter 1

Overview

The Einstein equations for the geometry of the relativistic gravitational field are intricate. To derive exact solutions, one must impose stringent symmetry assumptions. Despite these assumptions, several of the solutions obtained in this way are physically useful. Notable among these are the asymptotically flat, vacuum black holes: the Schwarzschild [319] and the Kerr [174] families of spacetimes. There appear to exist [82, 147] real objects, “astrophysical black holes”, which form naturally from the gravitational collapse of large stars [181, 278, 279], possessing many properties of these solutions [277].

At times, stars large enough to become astrophysical black holes form close enough together to be gravitationally bound, but far enough apart to avoid direct exchange of matter [112, 244]. The two bound stars in this case evolve independently, potentially collapsing into a pair of astrophysical black holes called a black hole binary (BBH) [83, 244]. This motivates the so-called “two-body problem in general relativity” [101], of finding exact solutions with two black holes. Except in extremely contrived situations (e.g. [134]), such systems admit no spacetime symmetries at all [83].

While exact BBH solutions are therefore not available, exact general relativity (GR) nevertheless gives some insight. Having a time-varying quadrupole moment, the binary will radiate gravitationally [216, 335], causing its constituents to draw closer and eventually to merge [68, 83]. To leading order the radiated power will be quintic in the inverse binary separation [335]. This is a much sharper scaling than that from other astrophysical effects such as dynamical friction or gas drag, and so the BBH should eventually decouple from its environment and be well-described by vacuum GR [274].

Further detail requires quantitative access to the evolution. There are essentially three approaches. In the Post-Newtonian expansion [56], relativistic corrections are added successively to the Newtonian theory. The resulting equations are most accurate when motion is slow. They are thus suitable for the description of well-separated BBHs during
early inspiral. Meanwhile, in the self-force formalism \[108\] the trajectory of one hole is computed perturbatively from the timelike geodesics of a Kerr spacetime appropriate to its counterpart. This is most accurate when the actual trajectories in fact resemble Kerr geodesics. Since the Kerr spacetime is a fully relativistic solution, speed and field strength are not directly relevant, but the size of the perturbation, and therefore the mass-ratio of the binary, is. Black hole binaries with very small mass-ratios are thought to occur naturally during, for example, mergers between supermassive and stellar black holes in galactic cores \[52, 274\], so this technique does have astrophysical motivation.

Finally, in numerical relativity (NR), the chief subject of this thesis, the Einstein field equations are solved directly by numerical integration. In principle any BBH solution could be obtained in this way, but in practice computational expense restricts attention to comparable-mass binaries with low-to-moderate spins \[207\] and few (10-50) orbits \[83, 131\]. Exactly which parameters are favourable is something of a moving target, as for example initial data improves \[232, 261\] to accommodate more extreme parameters. Recent work has included, for example, simulations with mass-ratios as high as 1:100 \[148, 200, 202, 222\], and an extremely long simulation covering 350 gravitational wave cycles \[291\]. Spin limitations imposed by initial data have been met \[99, 142, 254\], and even exceeded \[205, 208, 220\], by several studies, with recent work \[268\] approaching astrophysical spin limits \[304\].

The first published attempt at the numerical integration of a binary black hole spacetime was in 1964 - before the development of modern supercomputers - by Hahn and Lindquist \[141\]. Follow-up attempts over the next decade included increasingly realistic simulations of gravitational collapse \[212\], black-hole scattering of gravitational radiation \[315\], and further attempts at black hole binaries \[114, 285\]. Around 1977 it became clear that recently-proposed gravitational wave detectors would require accurate BBH simulations to improve sensitivity \[284\], leading to organized efforts at their production \[110, 284, 286\].

Nevertheless, it was not until 1993 \[33\] that a black hole binary was successfully simulated past merger. This merger was a head-on collision, which turns out to be much easier to simulate than a full inspiral. Stable simulations of generic mergers were made increasingly urgent by the waveform requirements of the gravitational wave detectors LIGO and VIRGO, which reached their full design sensitivities in 2005 \[11\] and 2009 \[12\]. Thus, in 1995 a coordinated effort, the Binary Black Hole Grand Challenge Project \[87, 211\], was initiated, leading to grazing collisions in 2000 \[27, 64\] and, rather suddenly in 2005, to long-term stable evolutions \[247, 248\]. Further long-term evolutions using the BSSN formulation and moving puncture approach, a qualitatively different set of methods
to those of Pretorius that has influenced many subsequent codes, were first performed in 2006 [44, 73, 74]. In 2015 the Advanced LIGO detector made its first detections of gravitational waves [6, 8, 9], with improvements to sensitivity and parameter estimation made possible by numerically generated waveforms.

At present, the most important application of NR remains the production of gravitational waveforms for comparison with data from ground-based gravitational wave detectors [17, 18, 90, 149, 171, 219]. NR codes have been so successful at producing these that they are sometimes viewed as merely waveform factories. We view this as unfortunate. NR is in fact a powerful theoretical tool, allowing access to dynamical spacetimes without any prerequisite symmetry.

Unfortunately, NR is hard. It is hard for computers, because useful integrations require 10000s of floating-point operations per gridpoint per timestep, translating to weeks or months of wallclock time per simulation in practice. It is hard for developers, because setting up the initial data, formulating the evolution equations appropriately, constructing the simulation domain, and controlling instabilities requires detailed, low-level programming. And it is hard to interpret, because the simulations adopt a particular spacetime gauge choice, such that extracting physically meaningful quantities can be complex.

In this thesis we take steps to make NR easier, and thus to expand its breadth. We begin, in Chapter 2, with a brief review of the field’s theoretical and technological basics. Next, in Chapter 3 we detail efforts to make NR simulations faster by running them on Graphics Processing Units (GPUs). Within waveform science, long wallclock times slow the production of template banks, impair follow-up simulations of observed compact binary mergers, and ultimately, conceal mergers in difficult-to-simulate regions of parameter space, such as at high mass-ratio, extreme spin, or nonzero eccentricity.

Outside of waveform science, theoretically interesting simulations tend to be computationally expensive. Comparisons with analytic approximation schemes, for example, involve simulations in parameter regimes where the analytic scheme works well, which is usually where NR does not. Interesting strong-field effects are most relevant to long, eccentric simulations at high mass ratio. Higher-dimensional spacetimes are computationally expensive because the extra dimensions result also in extra gridpoints [78]. And holographically-motivated spacetimes tend to have high resolution needs, due to instability to gravitational collapse from repeated scattering and refocusing of gravitational radiation off the AdS boundary [78].

In this thesis we consider a specific NR code, the Spectral Einstein Code SpEC [3]. We improve its speed by porting it to run on the GPU, an alternative processing architecture
to the more familiar CPU. GPUs can potentially outperform CPUs because they connect vastly more individual processors [226]. Attaining that potential, however, requires fine-tuned, low-level, imperative programming. This has mostly frustrated efforts to use GPUs for NR.

Our port avoids explicit GPU code as much as possible, instead relying on prepackaged libraries and the automatic porting framework described in Chapter 4. In Chapter 3 we describe our port in detail, along with benchmarks of each ported module, and of the overall code simulating an isolated black hole.

In Chapter 4 we present a software package designed to make NR easier for developers. Preparing a usefully-stable NR evolution involves a complex and tightly coupled battery of technologies to e.g. find apparent horizons, refine the grid, advance the timestep, and extract data. The sheer complexity of the resulting code makes modification, for example to perform new physical studies, a formidable task. At heart, however, the code mostly expresses operations upon mathematical tensors, which are represented numerically as multidimensional arrays. The “index Hell” of general relativity manifests as code like this:

\[
\begin{align*}
&\text{for (int } i = 0; i < 3; ++i) \\
&\quad \text{for (int } j = 0; j < 3; ++j) \\
&\quad \quad \text{for (int } k = j; k < 3; ++k) \\
&\quad \quad \quad \text{for (int } l = 0; l < 3; ++l) \\
&\quad \quad \quad \quad \text{Gamma}(i, j, k) += 0.5 \times g(i, l) \times (dg(j, l)(k) + dg(l, k)(j) - dg(j, k)(l));
\end{align*}
\]

which computes the Christoffel symbol of the second kind \text{Gamma} from the metric \text{g} and its first partial derivatives \text{dg}. We much prefer to write code like this:

\[
\begin{align*}
&\text{Gamma}(\text{Sym}<1,2>(), i, j, k) = \\
&\quad 0.5 \times \text{Sum}(\text{Sym}), \\
&\quad \quad g(i, j, k) \times (dg(j, l)(k) + dg(l, k)(j) - dg(j, k)(l));
\end{align*}
\]

In this thesis, we thus develop a C++ library, \text{TLoops}, to make the latter snippet compile and execute. Our library checks tensor expressions for mathematical consistency at compile time and uses a syntax close to familiar mathematical notation. Its primary application, however, is to simplify GPU porting, since \text{TLoops} expressions may also be used to output equivalent CPU or GPU code. By linking the automatically-generated code to a second compilation the expressions may be ported in this way with no additional effort. As demonstrated in Chapter 4, the resulting expressions make efficient use of the GPU at gridsizes relevant to \text{SpEC}. 
Chapter 5 helps develop NR as a physical tool from a scientific perspective. The orbital eccentricity of a BBH is strongly damped by the gravitational radiation reaction, and so numerical studies have prioritized the quasicircular case \[111,130,158,159,218,287\]. However, mounting astrophysical evidence (e.g. \[37,162,217\], and many other references compiled in the introduction to Chapter 5) suggests that binaries with significant eccentricity at merger may nevertheless occur. Without eccentric waveforms, such binaries are difficult to see \[98,163,164\]. But they are also disproportionately interesting: at least in the large mass-ratio case, eccentricity can lead to qualitatively new orbital dynamics \[20,186,203\].

In particular, perturbative studies of high mass-ratio inspirals have found orbital resonances, in which the beaming of gravitational radiation produces a sharp “kick” \[117,118\]. Since the detailed behaviour of the inspiral in this case depends upon the orbital phase at which the binary enters resonance \[260\], the resulting waveform cannot be predicted from system parameters alone. If this effect occurs for comparable-mass eccentric binaries, the paradigmatic matched filtering approach to detection would require significant modification.

We therefore perform a range of simulations with high spin, moderate mass-ratio, and moderate eccentricity. To detect resonances, we develop tools to extract the so-called “fundamental frequencies” - roughly the frequencies at which the inspiral oscillates in each of the spatial Boyer-Lindquist coordinates - from the simulation. Ratios of these frequencies are only weakly gauge-dependent, so we are able to use them to compare our results with analytic approximations (that make different gauge choices). We identify the “resonances” which occur when the frequencies are in integer ratio. When our simulations pass through resonance, we are unable to discern any effect upon the inspiral.

Finally, in the concluding Chapter 6, we review the content of the main chapters, and end by outlining how this research program might proceed further.
Chapter 2
Numerical Relativity Orientation

2.1 Introduction

Under the influence of gravity, the trajectories of test particles bend towards matter. That the trajectories are independent of the particles’ compositions (and thus of any force which couples to a charge) suggests gravitation might be described “geometrically”. Analogously, a line drawn on a ball will be curved by the ball, even if the pen is moved “straight”. Geometry in this sense is encoded in the Riemann tensor \( R^{a}_{bcd} \), which we would like to couple with some notion of “amount of matter”. The decoration \( (4) \) indicates that a tensor is four-dimensional, to distinguish it from three-dimensional equivalents which we will be interested in later.

We would like to avoid instituting a preferred coordinate system upon spacetime and so the quantity measuring “amount of matter” will need to transform “like a tensor”, i.e. in such a way that equations written in terms of it take the same form in all coordinate systems. The stress-energy tensor \( T_{ab} \) has this property, while also including special relativity’s energy-momentum four-vector as components. We would therefore like to relate it to \( (4) R^{a}_{bcd} \).

Obviously \( (4) R^{a}_{bcd} \) cannot be directly equated to \( T_{ab} \) because of the differing number of indices. Its unique contraction, the Ricci tensor \( (4) R_{ab} \), can be, e.g.

\[
(4) R_{ab} = -\kappa T_{ab},
\]

(2.1)

where \( \kappa \) is an unfixed coupling constant. These are the field equations of Einstein’s early “Entwurf” theory of gravity. The stress-energy tensor \( T_{ab} \) ought to have vanishing covariant divergence in order to preserve local energy-momentum conservation, but \( (4) R_{ab} \) need not. In the Entwurf theory one therefore assigns \( (4) R_{ab} \) vanishing divergence as
a coordinate condition. This gives coordinates a physical significance, albeit a rather abstract one. Full coordinate freedom can be recovered by subtracting the divergence from the left-hand side. Demanding that torsion vanish then leads to the Einstein field equations
\[
(4) R_{ab} + \left( \Lambda - \frac{1}{2} (4) R \right) g_{ab} = 8\pi T_{ab}.
\]

The “cosmological constant” \( \Lambda \) cannot be fixed by this argument since it does not affect the overall divergence. The coupling constant \( \kappa \) is set to \(-8\pi\) by demanding agreement with Newtonian physics when the spacetime metric \( g_{ab} \) and the Minkowski metric have almost the same norm.

### 2.2 3+1 Formulation

One typically derives exact solutions of Eq. (2.2) by imposing enough symmetry to integrate by hand. The result of this is a spacetime’s entire self-consistent history. This is somewhat at odds with other physical theories, which tend to be formulated as initial value problems, and with our daily experience, in which phenomena are apparently sequentially caused, rather than fixed by spacetime’s global structure. Indeed, the basic idea of experimental science - predictions are compared with results - presumes that our theories admit initial value formulations.

It is not obvious that such a formulation of Eq. (2.2) exists, since for example both sides depend non-linearly upon the metric. Strictly speaking the theory actually does not admit one, since solutions exist which contain causal pathologies such as singularities. One generally expects these to be unphysical, and in any case to be hidden safely behind event horizons [238,320], and so for practical purposes, given realistic equations specifying the behaviour of matter, Eq. (2.2) may be written as an initial value problem.

In such an initial value formulation [21,51,105,280], spacetime is foliated by a sequence of spacelike Cauchy surfaces\(^1\) \( \Sigma_t \) and “3+1” variables are defined. These variables describe the full spacetime in terms of temporally-local fields on the hypersurfaces.

Each \( \Sigma_t \) is assigned a monotonically increasing scalar label \( t \) (its subscript in our notation), the global time variable\(^2\). Distances within a given \( \Sigma_t \) are measured by its three-
metric $\gamma_{ij}$, from which connection coefficients and curvature tensors may be constructed in the usual way, except that the spatial geometry is of course Riemannian rather than Lorentzian. These intrinsic quantities may be expressed in any set of coordinates.

We also need quantities that describe the manner by which the $\Sigma_t$s are embedded in the full spacetime. Denote by $n^i$ the unit timelike normal vectors to the $\Sigma_t$, which are tangent to the worldlines of the “normal observers”. These can be used to construct projections of spacetime quantities into the $\Sigma_t$. Denoting the Lie derivative along normal observers as $\mathcal{L}_n$, the *extrinsic curvature* $K_{ij}$ determines how the spatial metric $\gamma_{ij}$ deforms in time

$$K_{ij} \equiv -\frac{1}{2} \mathcal{L}_n \gamma_{ij}. \quad (2.3)$$

The remaining quantities encode our remaining coordinate freedoms. As the global time variable advances by an increment $dt$, the proper time $\tau$ of the normal observers advances by $\alpha dt$, where the scalar $\alpha$ is called the *lapse function*. We are furthermore free to choose any spatial coordinate system for spacetime, which may move relative to the normal observers. This will be encoded in the *shift vector* $\beta^i$, which measures how far the normal observers diverge from lines of constant spatial coordinate between slices.

Both $\alpha$ and $\beta^i$ are coordinate choices. As tends to be true of coordinates, all choices are equivalent, but some are more equivalent than others. For example, under *geodesic slicing* $\alpha = 1$, $\beta^i = 0$, the worldlines of the normal observers are just timelike geodesics. This obviously simplifies interpretation, along with analytic calculation since terms proportional to $\beta^i$ vanish, but is usually unsuitable numerically. Being attractive, gravitation tends to focus timelike geodesics into caustics, where the forward evolution becomes many-valued and the simulation will crash. Caustics can be avoided by manipulating the value of $\beta^i$, which may even yield superluminal coordinate velocities if black holes are involved.

The *spacetime* line element may be written entirely in terms of the above spatial quantities

$$ds^2 = \alpha^2 dt^2 + \gamma_{ij}(dx^i + \beta^i dt)(dx^j + \beta^j dt). \quad (2.4)$$

By decomposing the Einstein equations, Eqs. (2.2), into components which are parallel and orthogonal to the $\Sigma_t$s, we may rewrite them in terms of the 3+1 variables as well. In vacuum, raising and lowering spatial indices with the spatial metric $\gamma_{ij}$, and using $T_{ij}$

---

even in Minkowski spacetime.
to denote the covariant derivative of $T$ with respect to $j$, one obtains

$$ R + K^2 - K_{ij}K^{ij} = 0, \quad (2.5) $$

$$ K^j_{ij} - K_{ji} = 0, \quad (2.6) $$

$$ \partial_i \gamma_{ij} = -2\alpha K_{ij} + \beta_{ji} + \beta_{ij}, \quad (2.7) $$

$$ \partial_t K_{ij} = -\alpha_{ij} + \alpha(R_{ij} - 2K_{ik}K^{k}_{\ j} + K \dot{K}_{ij}) + \beta^k K_{ij,k} + K_{ik} \beta_k^{\ j} + K_{kj} \beta_k^{\ i}. \quad (2.8) $$

This particular decomposition is called the ADM formulation of general relativity [39, 326, 327]. Its four constituents come in two groups. Eqs. (2.5) and (2.6) contain no time-derivatives and relate only information within a particular hypersurface. They are called constraint equations because they constrain what gravitational fields are self-consistent on a given $\Sigma_t$. Given a “photograph” of spacetime, the constraint equations would allow one to determine its authenticity.

Eqs. (2.7) and (2.8) relate information between hypersurfaces. They are called evolution equations because they determine how gravitational fields evolve from a given ancestor. Given a “photograph” of spacetime, the evolution equations would allow one to determine its successors.

One often draws an analogy with electromagnetism, which can be covariantly formulated in terms of the Faraday tensor $F_{\alpha\beta}$ in a manner similar to Eq. (2.2),

$$ F_{\alpha\beta;\gamma} = 0, \quad (2.9) $$

where vacuum, and flat spacetime, has been assumed.

We can write an initial value formulation of Eq. (2.9) by projecting the Faraday tensor onto a foliation of spacelike hypersurfaces. In vacuum on flat spacetime, we first define electric and magnetic field vectors, $E$ and $B$, in terms of inertial observers. The Cartesian components of these fields are given by $E_i = F_{0i}$ and $B_i = -\frac{1}{2}\epsilon_{ijk}F^{jk}$, where $\epsilon_{ijk}$ is the completely antisymmetric Levi-Civita symbol. The projections onto the foliation then result in two time-independent constraint equations

$$ \nabla \cdot B = 0, \quad (2.10) $$

$$ \nabla \cdot E = 0, \quad (2.11) $$

that determine which spatial field configurations are legitimate, along with two evolution
Chapter 2. Numerical Relativity Orientation

The evolution equations of both theories “preserve the constraints”. Using an exact solution to the constraints as initial data for the evolution equations results in a spacetime that satisfies the constraints at all times. This is true for any initial-value formulation of GH, with the crucial word being “exact”. Finite-precision numerical evolution will inevitably introduce small errors, both in the initial data and at each timestep. Eqs. (2.7) and (2.8) will amplify these errors exponentially in time. In practice this will cause a physical simulation to quickly crash, which is fortunate, since otherwise it would silently produce exponentially inaccurate results.

2.2.1 From an Initial Value Formulation to a Numerical Relativity Code

Well-Posedness and Reformulation of the Evolution Equations

We would like to obtain an initial value formulation which does not exponentially amplify floating-point errors. The difficulty is that Eqs. (2.7-2.8) are not well posed [78,138,140,156,221,251,252,265]. For an initial value problem in 1+1 with space coordinate $x$ and time coordinate $t$, well-posedness amounts to the existence of a norm $||\cdot||$ such that linear perturbations $\delta u$ about a solution $u_0(t,x)$ are bound by some function $F(t)$ of time only; e.g. [78,138]

$$||\delta u(t,x)|| \leq F(t)||\delta u(0,x)||.$$ (2.14)

Well-posedness is a necessary condition for numerical stability, though not sufficient, since $F(t)$ could be unacceptably steep. However, it is difficult to prove well-posedness directly, and thus one instead usually discusses the less restrictive concept of strong hyperbolicity [251,252,265]. For a first-order system in a set of variables $u(t,x)$

$$\partial_t u = P(t,x,u,\partial_x)u,$$ (2.15)

Note that the Maxwell equations are already stable. It thus took some time for the importance of a stable formulation to numerical simulation to be widely appreciated.
Chapter 2. Numerical Relativity Orientation

hyperbolicity (strong or not) in the case of constant coefficients is the demand that the smooth differential operator $P$ have only imaginary eigenvalues \[78\]. Strong hyperbolicity, which is a necessary condition for well-posedness \[297,298\], is the further demand that $P$ have a complete set of linearly independent eigenvectors. For fixed gauge, the ADM equations (2.5-2.8) are only weakly hyperbolic \[180,252\] (i.e. they meet the conditions for hyperbolicity, but not for strong hyperbolicity), and therefore are not well-posed. Any numerical evolution of them is therefore doomed to eventually (in practice, very quickly) crash.

The left-hand sides of the constraint equations (2.5) and (2.6) vanish for physical fields. Since arbitrary multiples of zero may be freely added to any component of a tensor, arbitrary multiples of the constraints may be added to the ADM equations (2.5-2.8) with no effect on the exact solutions. Perturbed solutions, on the other hand, may be affected dramatically. It is in this way possible to make the system numerically stable. This, along with changes of variables to isolate undesirable terms into fewer equations, is the essential principle behind reformulating the Einstein equations, although the practicalities are much more involved.

There are two genres of strongly hyperbolic reformulation in widespread use. The BSSN formulation \[50,281\] was developed and employed numerically about a decade \[78\] before its strong hyperbolicity was demonstrated \[138\]. The generalized harmonic formulation \[123,123,128,137,196,248\] is notable due to its use by Pretorius in 2005 to produce the first successful stable BBH evolution \[247,248\]. It is also used by the numerical relativity code SpEC, which is the main numerical tool used in this thesis. We therefore discuss it briefly here, although the original accounts \[196,248\] are comprehensive and accessible.

Early studies of gravitation \[113,119\] sometimes demanded that the coordinates $x^\alpha$ satisfy the scalar wave equation, $\Box x^\alpha = -g^{\mu\nu}\Gamma^\alpha_{\mu\nu} = 0$. In such coordinates, the harmonic gauge, the Ricci tensor becomes a scalar wave operator at second order and the Einstein equations become strongly hyperbolic. One possible approach to numerical relativity is therefore to simply impose harmonic gauge \[43,128,293,294\]. Such coordinates may, however, fail to cover the entire manifold, or may result in coordinate pathologies. Furthermore, imposing them exhausts general relativity’s gauge freedom, which could otherwise be put to better use.

One can achieve a hyperbolic representation of the Einstein equations for arbitrary coordinates \[123,128\] by assuming that the latter satisfy the scalar wave equation inhomogeneously. Thus, one introduces functions $H^\alpha$ of the coordinates and fields, and
assumes

\[ H^\alpha \equiv \Box x^\alpha = -g^{\mu\nu} \Gamma^\alpha_{\mu\nu}. \] (2.16)

This results in the same hyperbolicity properties as does harmonic gauge. Further multiples of the constraints may then be added to the resulting Einstein equations such that any short-wavelength deviations from the constraints are actually damped out rather than amplified [137]. Unlike harmonic gauge, the conditions of the generalized harmonic formulation do not exhaust any gauge freedom, since \( H^\alpha \) may themselves be freely specified, either by imposing extra evolution equations [248], or by setting them as a fixed algebraic function of the coordinates [196].

Discretization

To actually solve whatever strongly hyperbolic evolution equations we decide upon numerically, we must discretize them, replacing them with relations between finite arrays that approximate continuous solutions. There are two discretization schemes in common use within NR codes. In the conceptually simpler finite differencing schemes, one samples functions \( u(x) \) to be solved for at a discrete set of spacetime points \( x_i \) separated by a finite, and usually constant, coordinate distance \( \Delta x \). One then approximates derivatives using this finite (rather than infinitesimal) distance. More accurate approximations, which will also require more gridpoints per derivative to compute, can be found by comparing Taylor expansions about gridpoints.

In numerical relativity, the major alternative approaches to finite differencing are the spectral methods [60, 61, 75, 76, 122, 135, 179], to which SpEC owes its name. Here, one first chooses a complete set of basis functions \( \phi_k(x) \) on space. Next, one chooses a fixed set of \( N \) collocation points, and demands that at these points the evolution equations and any boundary conditions be exactly satisfied. The solution to the system is then approximated as a sum over the basis functions

\[ u^{(N)}(x) = \sum_{k=0}^{N-1} \tilde{u}^{(k)} \phi_k(x). \] (2.17)

When the solution is smooth and the simulation domain has a favourable topology, we can choose the basis functions and collocation points such that the \( \tilde{u}^{(k)} \) can be computed from the \( u_N \) in \( \mathcal{O}(N \ln N) \) operations. Having computed the \( \tilde{u}^{(k)} \) we can evaluate arbitrary derivatives of the \( u^{(N)} \), also in \( \mathcal{O}(N \ln N) \) operations. This results in exponentially [179] faster convergence at fixed accuracy than any finite difference method.

Some strings are attached. The fast convergence is only for favourable topologies
and choices of collocation points, which complicates adaptation to new problems. As currently used in NR, spectral methods are difficult to parallelize. They furthermore fail completely if the solution develops a discontinuity. Non-pathological discontinuities never form in vacuum gravity, but are ubiquitous to fluid simulations including e.g. shock waves. To simulate fluid systems, which notably include neutron star binaries, SpEC handles the fluid regions by finite differencing.

**Initial Conditions**

Before we can perform a simulation we must first construct initial conditions. We do not have space to describe this complex subject apart from some qualitative physical aspects, though many comprehensive reviews exist [21, 51, 94, 133]. For BBHs, we seek hypersurfaces of spacetimes with astrophysically interesting parameters (mass-ratio, spin, orbital frequency), analogous to those of a natural binary. Such surfaces must be self-consistent, which makes them difficult to construct.

The currently-standard approach is to construct an ansatz hypersurface with free parameters roughly analogous to those of astrophysical interest, feeding in target values as input, and relaxing the parameters towards consistency with the constraint equations. The result of this will be a binary black hole spacetime which, while a physical solution to the vacuum field equations, only approximates the target system because it is not in “quasiequilibrium”. The binary partners will be distorted, due essentially to our having chosen a hypersurface that implies an astrophysically unrealistic past history.

We are saved by the “no-hair” [262] behaviour of the binary constituents. We expect [80, 145, 170] eternally isolated black holes to be described by the Kerr family of solutions [174]. Departures from Kerr are corrected by powerful radiation whose back-reaction smooths any asymmetries (e.g. [95–97, 102, 315, 330]). If the hole is not too far from Kerr, the resulting waveform takes the “ringdown” form of a damped sinusoid.

In a BBH simulation the approach to Kerr occurs over two timescales. Over about two simulation light-crossing times, the horizons in the initial data surface emit their most egregious asymmetries during a burst of so-called “junk” radiation. The back-reaction of this radiation, which is characterized by too fine a length scale to be resolved in practice, causes an unpredictable “kick” to the system. Due to this kick, system parameters cannot be set, but only targeted. The practical importance of the junk-radiation kick depends to some extent on the system being simulated - absorption of junk tends for example to lower the spins of black holes, which is the leading reason that high-spin simulations are difficult.\(^4\)

---

\(^4\)We thank Josh Faber for highlighting this latter point.
To avoid contamination from junk-radiation kicks, we would need a means of constructing initial data hypersurfaces which were already in quasiequilibrium. These would need to include the mutual tidal distortion of the black holes, self-consistent initial velocities, scattered gravitational radiation from earlier phases of the evolution, etc, and it is not known how to do this. On the other hand, the same “no-hair” behaviour that causes the kick essentially guarantees we will end up with some astrophysically plausible BBH provided that the initial data at least contains horizons.

The second timescale is that of the merger itself. Since the horizons will eventually merge there is, strictly speaking, only one black hole in the spacetime, and it is not a Kerr solution. It is not even approximately Kerr, so the approach is not exponential and need not take the form of a ringdown, but the “two” holes must nevertheless eventually either merge or infinitely separate. The “resulting” gravitational radiation in the former case drives the inspiral. Post-merger a second ringdown occurs, driving the merged black hole exponentially towards Kerr.

### Singularity Avoidance

Interesting spacetimes will normally contain gravitational singularities. It is a basic assumption of the 3+1 approach that singularities do not occur, so this would seem to present a problem. In astrophysical situations, however, one expects singularities to be hidden by the cosmic censor [238, 320]. Thus, assuming interest in only in the external geometry, regions with singularities in their pasts may be safely ignored. While the full geometry might be of interest in non-astrophysical applications, such as holography, modelling it would in any case presumably require access to unknown quantum corrections.

The trick is therefore to somehow exclude the singular regions from the simulation. This can be done in several ways. For example, one could choose singularity-avoiding coordinates which do not cover the pathological regions of the manifold [22, 59, 115, 128]. This approach has two chief difficulties. First, it consumes gauge freedom. Second, it can lead to inefficiently small proper time intervals between near-horizon regions of the simulation hypersurfaces at late times. This last difficulty is chiefly historical, and can be dealt with by appropriate choices of the shift vector such that near-horizon observers are superluminal [26, 74, 313].

An alternative approach, which SpEC uses, is called excision [300]. The idea here is to find the horizons in the simulation and discard all data within [24, 46, 151, 269, 282, 288, 289]. Since locating the event horizon requires knowledge of the future, excision would be a hopeless approach if it were the surface to be excised. Instead, we find [23, 136, 271, 301, 302] apparent horizons (AH), which can be defined with knowledge of
Chapter 2. Numerical Relativity Orientation

a single spacelike hypersurface only, in terms of the following sequence of concepts. A *trapped surface* is a surface at which outgoing null geodesic congruences have non-positive expansion. If that expansion is zero, the surface is *marginally trapped*. An apparent horizon is the outermost marginally trapped surface in a sequence within a given spacelike hypersurface. Given cosmic censorship and appropriate energy conditions every apparent horizon is either coincident with or interior to an event horizon \([146,319]\). Note however that an event horizon can exist without an apparent horizon forming \([272,321]\)^5.

After the simulation is complete, the event horizon may be found if desired \([54,167]\) in post-processing, most efficiently by tracing by distant null geodesics backwards and finding surfaces at which they converge \([32,91,104,193]\). See \([303]\) and references therein for much more detail.

This approach has some difficulties as well. The apparent horizon must be located quite accurately, which entails some extra expense. The fluctuating excision boundary requires that new points need to be stably deleted and added to the simulation, which can be complicated \([25,65,93,247,283]\), especially for spectral methods \([269]\). On the other hand we retain our gauge freedom, and the locations of the apparent horizons are usually of interest anyway.

### 2.3 Applications of NR

Usually, the desired output from a numerical relativity simulation is the gravitational waveform at spatial infinity, which can be extracted via a number of techniques \([55]\). Aside from providing useful gauge-invariants, such waveforms are of concrete significance to gravitational wave astronomy \([243,266]\), particularly with regard to ground-based detectors \([4]\) such as Advanced LIGO \([5,49,144]\) and VIRGO \([13–15]\). Such detectors control noise by comparing observation with theoretically modelled waveforms in a technique called *matched filtering*. This technique, instrumental to the actual detection of gravitational waves \([7–9]\), requires a “template bank” (e.g. \([17,19,66]\)) of model waveforms which can be mapped to system parameters. While numerical relativity is too computationally expensive to generate the whole template bank itself, numerically generated

---

5These definitions can be illustrated by the following cartoon. At a trapped surface, light from a flashlight will fail to shine outward, no matter which way the flashlight points. At a marginally trapped surface the light will fail to fall inward either, if the flashlight is pointed directly away from the surface. Typically, trapped surfaces will occur in a nested sequence. The apparent horizon is the outermost of these. At the event horizon, light is doomed to cease outward propagation eventually. But it may initially propagate outward before for example being swallowed by an expanding black hole; indeed the region about an event horizon may even be flat. Thus, the apparent horizon lies inside or upon the event horizon.
waveform catalogues (e.g. [219]) are used to calibrate [16, 40, 41, 70, 171, 236] waveform models in order to better model the strong-field regime, resulting in greatly improved sensitivity.

Astrophysically [166, 191], NR has improved qualitative understanding of strong-field comparable-mass BBH dynamics, by essentially confirming the rather prosaic picture of the semi-analytic effective one-body approach [69, 100]. Comparable-mass quasicircular (circular up to the damping effects of gravitational radiation) black hole binaries smoothly increase in orbital frequency during inspiral, release a powerful, but still smooth burst of gravitational radiation through merger, and then are quickly damped towards a Kerr solution. The post-merger black hole may, however, inherit a considerable velocity, typically about 175 km/s but sometimes much higher [45, 132, 154, 201]. This can be sufficient to escape globular clusters and potentially even galaxies. Binaries with significant eccentricities and highly unequal masses can be shown using perturbative methods to exhibit more complex behaviour, including potentially sensitive dependence upon initial conditions [117, 118], though this cannot yet be confirmed experimentally since existing gravitational-wave detectors are sensitive only to comparable-mass binaries. Our preliminary study [192] of eccentric comparable-mass binaries failed to find such sensitive dependence.

NR has been and continues to be an important tool for the study of non-vacuum binaries [107, 242] and for core-collapse supernovae [234]. These simulations are exceedingly difficult because of the complex and poorly-understood behaviour of the relevant matter fields. Nevertheless, these are extremely active topics which we lack the space to review. NR has also found some sporadic application in cosmological settings [31, 129, 153, 317, 318], though situations requiring fully nonlinear, but still classical, dynamics are rare in that field.

NR can also be useful to investigations of relativity theory itself. Perhaps most famous are the “critical collapse” results of Choptuik [85] and of the many subsequent studies reviewed in [139]. These follow the collapse of a family of initial data varying by a single parameter \( p \), under various symmetry and stress-energy assumptions (Choptuik’s original study for example considers a massless scalar field in spherical symmetry). In many such situations there is a critical parameter \( p_* \) such that for \( p > p_* \) only, the late-time solution is a black hole with mass \( M \propto (p - p_*)^\gamma \), where the critical exponent \( \gamma \) is independent of at least some aspects of the initial setup. For \( p = p_* \) the (unstable) critical spacetime is found to be self-similar. This behaviour is reminiscent of a type II phase transition, suggesting connections with conformal field theory and perhaps with some sort of spacetime microphysics. It is probably safe to say that the significance of
such connections remains an open question.

The critical spacetime can also be shown to contain a naked singularity (a gravitational singularity which is not enclosed by an event horizon), which historically forced a modification to the cosmic censorship conjecture that naked singularities do not occur “under physical conditions”. The basic idea of cosmic censorship is left unharmed by existing critical collapse results since the naked singularity is unstable [89]. Still, this brush with death might lead one to wonder to what extent cosmic censorship is an artifact of the symmetry assumptions necessary to analytic work in relativity.

In fact the reverse appears to be true: naked singularities are an unstable and degenerate phenomenon of unphysical symmetry. For example, naked singularities can form from the collapse of an ideal fluid [172], but only when that collapse is somehow fine-tuned. A possible counter-example may have been found for black strings in 5 dimensions [190], which in cylindrically symmetric simulations possibly achieve unbounded curvature in finite time, though this remains somewhat unclear.

NR has also been used to test (and support) [86, 103, 109, 255, 328, 329] Kip Thorne’s “hoop conjecture” [305] that a black hole will form whenever a mass $M$ is contained within a radius $R/2$ in all directions. Investigations of for example alternative gravity theories, higher-dimensional black-hole topologies, the stability of spacetime especially including AdS, and holographic correspondences more generally remain ongoing; see for example [78, 331].

We understand the studies presented in this thesis as a preliminary contribution to NR’s use as a theoretical tool. This use is limited by computational expense, code complexity, and by the scientific difficulties of constructing and interpreting new simulations. Therefore, we present in the ensuing chapters work towards making the NR code SpEC faster and easier to work with, followed by an actual simulation investigating the dynamics of comparable-mass eccentric black hole binaries. Of course these studies are relevant to waveform science as well. With the recent observations of gravitational waves by Advanced LIGO on the one hand, and the rapidly increasing importance of higher dimensional gravitational physics to for example holography on the other, we expect NR’s role on both fronts to dramatically expand.
Chapter 3

GPU-Accelerated Simulations of Isolated Black Holes

3.1 Introduction

Numerical relativity (NR), the direct numerical integration of the Einstein field equations, is now a mature subfield of computational physics, owing largely to its key contributions to gravitational wave observations by ground-based detectors [7–9]. Detailed knowledge of expected waveforms, themselves coming ultimately from NR simulations, are required by these detectors to maximize sensitivities, to interpret observation, and to make tests of general relativity [92]. Ground based detectors’ relative insensitivity to e.g. eccentric binaries is, conversely, in part due to a lack of production-quality simulations in the eccentric region of parameter space [192], a situation which also impairs comparisons with analytic theory.

The intricacy of the Einstein equations presents two challenges to NR. First, interesting simulations are expensive, with wallclock times measured in weeks or months. Second, codes able to perform such simulations are quite intricate from a software engineering perspective. Applying them to new regions of the binary black hole parameter space - let alone to new spacetimes - can require months of effort by small groups of experts. These issues are difficult to address simultaneously, since improving runtime tends to complicate code, and vice versa.

Twenty years ago, the simplest solution would have been to simply wait for hardware to improve. Unfortunately CPU clock frequencies have been essentially static for some time now, with new high-performance computers instead employing increasingly massive levels of parallelism. But few codes scale to 100000s of CPUs without considerable
reformulation.

Easier speedups can sometimes be achieved using “accelerated” architectures. A popular choice is the Graphics Processing Unit (GPU). Problems in graphical computation might involve, for example, computing pixel states as a virtual object moves three-dimensionally. Each pixel is data independent, and the fundamental operations are linear transformations such as rotations. Thus, viewed more abstractly than perhaps originally intended, GPUs are optimized for highly parallelizable linear operations. These are performed by slow, but numerous and tightly-coupled, processors connected by various hierarchies of memory. The tight coupling and fast RAM make intra-GPU communication inexpensive. Because of this, GPUs are, for suitable problems, potentially dramatically superior to CPUs in terms of e.g. FLOPS-per-watt.

The extra parallel cores replace the CPU’s extensive control circuitry, which for example rearranges instructions to optimize single-thread performance, along with much of its cache memory, which increases the speed of non-contiguous memory access. GPUs are thus less flexible than CPUs, and less able to handle fundamentally serial problems. But even for less-than-ideal use-cases, GPUs have a major advantage over CPUs: they continue to demonstrate Moore’s-law-like FLOPS/year scaling with new releases. Therefore, once the initial investment is made to produce a port, further speedups can be made by simply buying new hardware\footnote{The speedup-over-time is largely due to newer cards supporting yet-higher levels of parallelism. Problems that already exhaust the parallelism of an existing card will not benefit from the scaling, or will benefit only weakly.}.

GPUs now enjoy widespread and increasing use as “accelerators” of numerically-intensive, linear-algebra-heavy tasks such as physical simulation and deep machine-learning. At the time of writing, they have not been widely adopted in NR, due likely to the complexity introduced by the low-level nature of GPU coding. Some previous applications of GPU computing to NR do, however, exist. Zink [333] used a CUDA-based finite-difference code to evolve a gauge wave upon Minkowski spacetime. Later, he developed HORIZON [334], a GPU-accelerated GRMHD code. Yang et al. [324] simulated plunging black-hole binaries in the BSSN formalism using a finite-difference CUDA port of the numerical relativity code AMSS-NCKU [77, 126] in concert with an AMR-like algorithm of their own design. Chen [84] used a GPU-accelerated approach to solve sample coupled elliptic equations using the spectral collocation and spectral Galerkin methods. The Teukolsky master equations [299] describing perturbations to the Kerr spacetime have been solved using the Cell processor SDK [176], OpenCL [88, 178], and CUDA [177]. Herrmann et. al [155] used CUDA to integrate the PN equations at an unspecified
truncation order. Brugmann [67] developed a GPU algorithm to integrate PDE systems relating time-dependent tensor fields on a spherical shell using pseudospectral methods. Perhaps most similarly to our work, automatically-generated GPU code has been used to benchmark binary black holes inspirals simulated using the Einstein Toolkit [57].

In the present work we describe our approach to the GPU porting of a specific NR code, the Spectral Einstein Code (SpEC) [3]. Multi-domain spectral methods like those used by SpEC solve PDEs by dividing the simulation volume into “domains” upon which the solution is smooth. Within each domain, the solution can then be represented as weights to a truncated sum of basis functions. Data thus represented are nonlocal in space. Every processor working on the same domain will generally require access to the full spectrum.

SpEC already uses MPI to assign each domain to a different processor. In setups where multiple CPU cores share RAM, it should also be possible to assign multiple cores to a single domain, but this has not been achieved in practice despite repeated efforts. On the other hand, the many processors within a single GPU also share a unified pool of memory. Access to this memory is sufficiently fast that the entire GPU can work on the same domain. GPUs, in other words, enable SpEC to scale to higher levels of parallelism than would be otherwise possible. Conversely, SpEC’s inability to utilize OpenMP means that the relevant benchmark in this work is the performance of a single CPU to an entire GPU.

In principle the Einstein equations are merely a specific example of a hyperbolic PDE system to be solved, which presents challenges on either the CPU or GPU identical to any other. The algorithmic reasoning employed in this work is indeed quite straightforward, and our challenges have instead been more practical. In simple terms, the SpEC source code is very long and complicated. Computational effort is spent mostly on a relatively small number of modules, but in practice these are each implemented by a complex and diverse set of subclasses depending on, for example, the topology of the domain (cubes, cylinders, spherical shells, etc.) upon which they operate.

Producing GPU equivalents of all the necessary instances would involve considerable effort. More problematically, upon completion code maintenance would become infeasibly difficult, since consistency between the CPU and GPU code bases would have to be continuously maintained at each revision. Of course, only relatively little code is actually performance-critical enough to yield practical benefits from GPU acceleration. Porting only these critical segments, however, results in numerous expensive CPU-GPU memory synchronizations, since the input to and output from the critical modules must be accessible to the relevant processor. This expense swamps any speedup in practice.
To keep the amount of redundant code manageable, we use a combination of porting strategies relying upon various levels of automation. At the highest level of automation, we have developed a C++ library, TLoops, the subject of Chapter 4, which provides technology to write spatially-decoupled tensor-algebraic expressions directly into C++ source code. For example, the TLoops expression in Listing 3.1 yields output equivalent to that in Listing 3.2.

Listing 3.1: Example TLoops expression.

```cpp
Tensor<DataMesh> dtg, K, db;
DataMesh alpha;
// initialize dtg, K, db, alpha
dtg(Sym<0,1>(), i_, j_) = -2*alpha*K(i_, j_) + db(i_, j_) + db(j_, i_);
```

Listing 3.2: C-style code, equivalent to Listing 3.1.

```cpp
Tensor<DataMesh> dtg, K, db;
DataMesh alpha;
// initialize dtg, K, db, alpha
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j <= i; ++j) {
        for (int a = 0; a < N; ++a) {
            dtg(i, j)[a] = -2.0*alpha[a]*K(i, j)[a] + db(i, j)[a] + db(j, i)[a]; // *
        }
    }
}
```

Here DataMesh is SpEC’s array class, representing one double precision value at each point on a simulation domain, while Tensor<DataMesh> is a container class representing one DataMesh for each component of a tensor (field). TLoops can run the code in Listing 3.1 at once, or can output equivalent GPU code to be linked against a separate SpEC compilation, allowing for efficient GPU porting with very minimal effort (the replacement of code in the form of Listing 3.2 with TLoops expressions such as those of Listing 3.1). TLoops will also output equivalent GPU code to the line marked `\*` in Listing 3.2, allowing almost the entire SpEC code to be (inefficiently) ported at once.

TLoops, thus, allows SpEC to keep data always on the GPU without any additional coding. Code segments which consume large amounts of wallclock time and which consist mostly of tensor manipulations, such as the code SpEC uses to advance the Einstein
equations in their generalized harmonic formulation by a timestep, can usually be sped up by at least 10 times relative to the GPU through the use of TLoops statements such as Listing 3.1.

At the next levels of automation, there are a number of modules that are performance-critical, but which cannot be handled by TLoops because they are not spatially-decoupled (i.e. their output at a given gridpoint depends on simulation data at other gridpoints). These modules, which include the differentiator for example, turn out to have several key features in common. First, each may represent any of various transformations, and often multiple implementations of each. The differentiator, for example, may be handled using a matrix multiplication, or by spectral methods whose details depend on the domain topology, with the choice being made by the user at runtime. The number of possible execution branches is too large to port everything by hand.

Fortunately, all such modules used by SpEC turn out to represent finite linear transformations. Furthermore, while the number of possible execution branches is very large, the number of actual branches encountered by a particular process is in practice manageably small. We therefore write GPU code which implements finite linear transformations, given an explicit matrix representation of them. When a new transformation is encountered, we feed delta-function input through the existing CPU code in order to trace out such a representation. We then cache the matrix so obtained and look it up whenever necessary. In this way large volumes of code may be ported with relatively little effort.

There are, finally, some modules which are neither amenable to TLoops nor to the cached linear transformation approach. This last class includes, for example, sequences of contractions with Jacobian matrices designed to transform the spatial coordinates of a spacetime tensor while leaving the temporal coordinates unchanged. These are few enough to simply port by hand.

Our strategy thus consists first of automated porting of existing expressions using TLoops, with the loops in some especially important modules replaced by TLoops expressions. Next, we port linear transformations by tracing out their explicit matrix representation, and port the few remaining important segments by hand.

The rest of this chapter is structured as follows. Section 2 describes SpEC and introduces the GPU architecture. Section 3 gives a detailed, module-by-module description of our port including benchmarks. Section 4 presents and discusses holistic benchmarks of the entire SingleBH test case. Finally, Section 5 draws conclusions and motivates future research.
Chapter 3. GPU-Accelerated Simulations of Isolated Black Holes

3.2 Technological Overview

3.2.1 The Spectral Einstein Code

The Spectral Einstein Code (\texttt{SpEC}) [3] is a C++ code designed to solve Einstein’s equations of general relativity. Its primary purpose is to simulate inspiraling and colliding black holes and neutron stars.

For a binary black hole spacetime, \texttt{SpEC} employs a domain-decomposition, dividing the computational domain into about 60 elements, or “domains”. Different domains may have different shapes, such as cubes, cylinders, and spheres. These may in turn have different connectivity and consequently different spectral basis-functions. In this paper, we only consider single black hole spacetimes, where the domain-decomposition consists of a set of concentric spherical shells.

\texttt{SpEC} employs the method of lines to evolve collocation point values of about 50 fundamental variables using a high-order Dormand Prince timestepper [246]. Non-linear terms such as those in the Einstein equations are directly computed at the collocation points.

Derivatives, filtering, and interpolation are performed using spectral transforms. First, the collocation data is transformed to the appropriate spectral space. Derivatives and filtering are then implemented as operations on the spectral coefficients. The result is transformed back to collocation space.

The evolved variables are tensorial, e.g. $\psi_{ab}(x^i)$. Here, $a, b = 0, 1, 2, 3$, indicate space-time components, and $x^i$ are the spatial coordinates. Some operations, like the computation of derivatives, operate on each tensor-component separately. Others couple different tensor-components. For instance, filtering in a spherical shell relies on a representation of data in terms of tensor spherical harmonics, to achieve a consistent truncation of components in angular resolution.

\texttt{SpEC} employs the dual-frame approach [269]. Here, data is represented at collocation points at fixed grid-coordinates, the coordinate system in which the domain-decomposition is specified. The evolution equations, however, are formulated in asymptotic inertial coordinates. The two coordinate systems share the same time-coordinate, and their spatial coordinates are related by a time-dependent coordinate transformation.

\texttt{SpEC} is a highly configurable code. Many modules are defined through abstract base-classes, and are implemented in derived classes. The concrete derived classes to be used for a certain domain can be chosen at runtime through parameter files. These choices include the coordinate mappings between simulation and output coordinates, which filters to implement, and how spectral transformations are performed (e.g. via FFT's or via a
BLAS-matrix call), and how interpolation is performed (via a direct summation of the spectral series, or via a FFT onto a finer grid followed by polynomial interpolation). This configurability leads to many execution paths through a program.

### 3.2.2 Programming for NVIDIA GPUs

Here we briefly introduce the pertinent characteristics of the GPU architecture [182, 224, 229–231], which is contrasted with the CPU architecture in Figure 3.1. The CPU employs a small number (1-4 in the diagram, and up to several 10s in contemporary examples) of cores that perform actual computations. All memory is accessible by all cores. Some, much faster, memory is used to cache data on the grounds that repeated accesses are likely. This caching is not managed explicitly by the user at the software level.

A relatively large amount of space is devoted to control circuitry, which for example performs hardware-level optimizations. The extensive control circuitry, transparent memory caching, low levels of parallelism, and fast serial performance give the CPU great flexibility to handle a wide variety of problems. Partly because of this flexibility, and partly because serial programs are comparatively easy to optimize at the compiler level, it is rare that a developer need think about the hardware when writing code.

The idea of the GPU is essentially to gut this entire structure and replace it with as many processing cores as possible. All such cores are connected to a pool of “global” memory, which is relatively slow, though still faster than CPU memory. A very small global memory “L2” cache may be present, but it is in practice negligible compared to the CPU cache, and thus not represented in Figure 3.1. Programmers must thus carefully manage the manner in which global memory is accessed to achieve reasonable levels of performance, especially since the speed of global memory access is very often the limiting performance factor.

The cores are organized into groups called streaming multiprocessors (SMs), represented in the diagram by horizontal bands of cores. Each SM (pairs of SMs in some architectures) houses a small amount of control circuitry and a hierarchy of much faster memory pools available only to it, represented by the two vertically stacked rectangles to the left of each SM in the diagram. These include another small cache, a pool of fast “shared” memory, and a number of very fast registers. A thread scheduler delegates processes to individual SMs.

Each SM has an independent control-logic, shared by all cores within each. This saves dye-space (c.f. Figure 3.1), but means that all cores in an SM execute instructions in lockstep. Each core may, however, execute these instructions upon a different
global memory address. GPUs are therefore designed to implement the SIMD (Single Instruction, Multiple Data) model of parallelism. Programmers must take care to avoid conditional statements which make the instructions executed by individual cores dependent upon their individual data. Such statements cause the entire SM to execute each branch of the conditional in serial.

For most of their history GPUs could be programmed only via specialized “shader” languages which were heavily oriented towards graphics tasks. Much more flexible access is now possible via generalized GPU frameworks such as NVIDIA’s own CUDA [226]. CUDA allows the GPU to be manipulated through a low-level C-like interface. Parallelism is abstracted as a hierarchy of logical structures adapted for execution by the physical structures described above.

These logical structures derive their names from an inconsistent and confusing metaphor with looms. Instructions are given to the GPU by writing a CUDA kernel, which is roughly analogous to a C function. Typically, a kernel reads an array from global memory, performs some computation, then stores the results back in global memory. When executing a kernel, the programmer assigns a number of blocks, and to each block a number of threads. Blocks are always local to an SM and may therefore exploit SM-local resources such as shared memory. Each individual thread, which has a unique index, serially executes the instructions in the kernel. The thread index can be used to address global memory, and in this way operations on arrays can be parallelized.

When the kernel is executed, the thread scheduler assigns the blocks to individual SMs. The blocks are then divided into groups of 32 threads called warps, which execute instructions in lockstep\(^2\). Each SM can simultaneously execute multiple warps, with the exact number depending on the specific GPU architecture. This allows SMs to hide latency: while an instruction in a particular warp e.g. waits for data, the SM may execute instructions from another warp rather than simply idling.

Writing efficient CUDA code requires low-level awareness of hardware. The largest performance issue comes from the fact that the CPU and GPU have physically different memory. Any data the GPU (CPU) needs from the CPU (GPU), including the kernel machine code itself, must be transferred over an interconnect. Such transfers must be kept to a minimum, both in size (since CPU-GPU interconnects are slow) and in number (since initiating a new transfer carries significant latency, and since all potentially-asynchronous GPU operations must be halted while memory is modified).

---

\(^2\)In an actual loom, the term “warp” refers to a group of threads which are drawn through a “weft” of perpendicular threads held under tension to make cloth. To our knowledge the terms “kernel”, “block”, and “grid” are not relevant to the textile industry.
Chapter 3. GPU-Accelerated Simulations of Isolated Black Holes

Figure 3.1: Abstraction (image from [226]) of CPU vs GPU architectures emphasizing differences between each. GPUs have vastly less control circuitry and many more processors (ALUs). GPU memory is arranged hierarchically, with a large “global” pool available to all processors and much slower “shared” pools available to individual “streaming multiprocessors” (the horizontal bands of cores).

The necessity of reasoning explicitly about the low level details of memory access accounts for much of GPU programming’s difficult reputation. A warp always accesses global memory contiguously as a unit. Anything other than contiguous accesses in groups of 32 entries requires that the entire warp make multiple accesses. The shared memory and registers localized within streaming multiprocessors permit comparatively fast random access, so data can be cached here after a contiguous access to global memory. But this must be done manually by the programmer, and the use of shared memory in particular can easily create e.g. race-conditions. The explicit synchronizations required to manage these, and the rather cryptic error messages supplied by CUDA when such management has been done improperly, can greatly complicate kernels.

At the kernel level the most important optimizations stem from two essential differences between the CPU and the GPU. First, GPUs suffer considerable single-thread latency. However, these latencies can potentially be hidden by asynchronous execution. Ideally, then, kernels and blocks should execute for times much longer than their scheduling overhead. They should also be numerous enough that all SMs are constantly occupied (but see [316]), which also helps to hide any remaining latency. Blocks should finally have threadsizes which are multiples of 32 (the number of threads in a warp), since warps are indivisible and otherwise some cores will be left idle.

Second, compared to the CPU, the GPU performs (parallel) computations far more quickly than memory accesses. A GPU-friendly algorithm should ideally be compute-bound, meaning that its arithmetic intensity, or ratio of computations to memory accesses, is sufficiently high that the algorithm becomes faster with increased computa-
tional power, rather than memory speed. In the opposite situation of a *memory-bound* kernel, speedup will be limited by the bandwidth of the GPU memory. This will be an important consideration when analyzing the performance of our implementations.

### 3.2.3 GPU Benchmarking

Throughout this study we will often be concerned with the actual performance achieved by GPU implementations of some algorithm compared with what is theoretically possible. We will also be concerned with the advantage achieved by the GPU, relative to the CPU, when used with hardware that is realistically available to SpEC users. In this subsection we lay out our general approach to benchmarking, and thus to quantification of such notions.

We first estimate the potential performance of each algorithm according to the following (rather well-established) framework. We view each algorithm as consisting first of $M$ memory transactions, i.e. loads and stores of size $w$ bytes to and from RAM, resulting in $D = 10^{-9}wM$ GB of data transacted total. All operations in this study are in double precision and we take $w$ to be 8 bytes throughout. The algorithm also performs $F$ floating-point operations - multiplications and additions - a term we loosely identify with “instructions”, and which we measure in GFLOPs. Such an algorithm can be characterized by its arithmetic intensity $I$

$$I \equiv \frac{F}{M}$$

which, since GFLOPs are just numbers, is dimensionless.

These $D$ and $F$ are what we estimate to be the *minimum* possible amounts of transacted data and floating-point operations that any implementation of a given algorithm must perform. We thus suppose that, if run on hardware which can process data at an optimal bandwidth of $B_i$ GB/s and an optimal processing power of $P_i$ GFLOP/s, an algorithm will ideally spend $D/B_i$ seconds on memory transactions and $MI/P_i$ seconds on floating-point operations. Assuming all latency can be hidden, compared with another device with optimal bandwidth and processing power $B_j$ and $P_j$ the potential speedup is then

$$\text{speedup} = \left( \frac{wB_i^{-1} + IP_i^{-1}}{wB_j^{-1} + IP_j^{-1}} \right)$$

which assumes that an implementation which minimizes $M$ and $F$ has been achieved on *both* devices.

When $I = I_{eq} \equiv wP/B$, a processor will spend equal amounts of time on memory
operations and arithmetic. Put differently, an algorithmic redesign that makes unnecessary the transmission of $D_-$ GB of data by performing $F_+$ extra GFLOPs will be an optimization when $F_+ < I_{eq} D_-$. We therefore consider an algorithm to be “memory” rather than “compute” bound when $I < I_{eq}$.

Depending on whether we expect algorithms to be memory or compute bound, we present benchmarks in terms of one of two quantities: the effective bandwidth $BW_{eff}$, measured in GB/s or the effective processing rate $P_{eff}$, measured in GFLOP/s. Given the actual measured time $t$ required to perform the benchmarked operation, these are defined by

\[
BW_{eff} \equiv \frac{D}{t}, \quad (3.3)
\]
\[
P_{eff} \equiv \frac{F}{t}. \quad (3.4)
\]

For algorithms which are heavily memory or compute bound, these quantities will be comparable to and bound from above by $B$ and $P$ for a given device, which allows for quick characterization of the achieved performance. More precisely, we can compute theoretically optimal performance metrics as

\[
BW_{eff, opt} = w \frac{BP}{wP + IB}, \quad (3.5)
\]
\[
P_{eff, opt} = I \frac{BP}{wP + IB}. \quad (3.6)
\]

Observed performance far beneath these values indicates that further attention to optimization may be worthwhile: the algorithm as written may, for example, be performing extraneous operations or spending excessive time on latency.

Table 3.1 shows the vendor-reported $B$ and $P$ along with an actual measured value for $B$ obtained by running the CUDA SDK program `bandwidthTest`, which simply times the result of a device-to-device memory transfer. Using these measured figures we also compute $I_{eq}$ for each of four devices: a single core of an Intel Xeon E5-2620 CPU, along with M2090, K80, and P100 GPUs. We use a single CPU core because, as discussed earlier, SpEC is incapable of OpenMP-style parallelization of work upon a single domain across CPU cores. In Figure 3.2 we furthermore display $BW_{eff}$ and $P_{eff}$ for each processor. We also plot the “speedup”, i.e. the ratio between the execution time on one of the GPUs from Table 3.1 with that of the CPU. Larger speedups indicate better GPU-than-CPU performance. We will be interested in this quantity (computed using the actually measured execution times) throughout this study.
### Table 3.1: Performance specifications for our benchmarked processors.

The columns are defined during the discussion in Section 4.5. “CPU” refers to a single core of an Intel Xeon (Sandy Bridge) E5-2620. The K80 actually contains two separate GPUs (which share memory) on the same card. Using both requires similar extra effort as multi-GPU programming generally, so we profile only one throughout. The K80 and P100 are also potentially capable of “GPU Boost”, which dynamically adjusts the core clock frequency if it is possible to do so without exceeding thermal and power limits (the CPU has similar capabilities). The “measured” bandwidths were obtained by running the CUDA sample program `bandwidthTest`.

<table>
<thead>
<tr>
<th>Device</th>
<th>B</th>
<th>P</th>
<th>( I_{eq} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>42.7</td>
<td>-</td>
<td>8.0</td>
</tr>
<tr>
<td>M2090</td>
<td>177.6</td>
<td>123</td>
<td>665.5</td>
</tr>
<tr>
<td>K80 (one card)</td>
<td>280</td>
<td>170</td>
<td>932–1456</td>
</tr>
<tr>
<td>P100</td>
<td>720</td>
<td>449</td>
<td>4036–4670</td>
</tr>
</tbody>
</table>

These figures allow us to draw two immediate conclusions. First, the theoretical speedups range between about 4 and 400, which given that NR runtimes are typically measured in weeks or months represents a dramatic increase in productivity even in the pessimistic case. Note that actually achieved speedups may in fact be higher than “optimal”, since the CPU algorithm may not be fully optimized, since one will typically rewrite an algorithm to achieve a more favourable value of \( I \) during a port, and since latency may not affect all processors equally in practice. Second, realizing such speedups requires high values of \( I \), to a much greater or even opposite degree as would be optimal on the CPU, as demonstrated by the approximately linear dependence of speedup upon \( I \) between \( I \) of around 1 to 100. Effective porting thus often requires implementations, and sometimes whole algorithms, to be redesigned in order to minimize memory transactions relative to floating-point operations.

In Figure 3.3 we, as an example, show benchmarking information collected from the P100 GPU performing the DiffJac operation described in Section 3.3.3. We measure performance by \( BW_{eff} \), computed from Eq. (3.3). This equation requires an estimate of the logical size of the operation \( D \), which for us is just 8 bytes multiplied by \( M \), computed also in Section 3.3.3. Higher values of \( BW_{eff} \) indicate better performance. We consider performance to be “good” when it is “near” the estimated optimal performance \( BW_{eff, opt} \) calculated from Eq. (3.5), and plotted in Figure 3.2.

Eq. (3.5) involves \( I \) and thus both \( M \) and \( F \). Therefore, our benchmark presentations will usually take the following form. First, we will introduce the operation to be
Figure 3.2: **Top Panels:** Effective bandwidth, processing rate, and theoretical speedup vs. a single core of the Intel Xeon E5-2620 as a function of arithmetic intensity in double precision for the three GPUs we benchmark in this study. Improvements to arithmetic intensity are critical around $I = 1 - 100$, when the speedup dependence is nearly linear. This assumes zero latency on both CPU and GPU, and that the algorithm running on both be exactly identical; neither assumption will hold in practice. The ranges account for the dynamical clock frequencies of all devices except the M2090. **Bottom Panels:** Zoom-ins around $I = 0$ to 5, relevant to the Jacobian contractions benchmarked later.
Figure 3.3: Individual raw benchmarks from the DiffJac operation (dotted coloured lines) on the P100 operating on $T_{abb}$. The median at each gridsize, which clearly tracks the overall trend, is overlaid as a dashed black line.

The machines we had access to for our K80 and P100 tests are head nodes, whose operating systems do not employ batched processing. As a result our benchmarks in these cases are much noisier than for the M2090 and CPU tests. We presume the noise to be due to machine resources being assigned to processes besides those we seek to time. To correct for this we run each test 50 separate times. The individual benchmarks show an obvious trend contaminated by rare, but extreme, dips in performance that do not persist across runs. We thus take, at each gridsize, the median result of the 50 runs. To illustrate this, Figure 3.3 plots 10 individual benchmarks from the DiffJac operation described in Section 3.3.3, with the median overlaid on top. The individual benchmarks mostly agree apart from isolated large spikes. The median tracks the agreement. The qualitative shape of this curve is discussed in Section 3.3.3.
3.3 Details of our port

3.3.1 Overview

We now turn our attention to SpEC-based black hole simulations. The case we consider throughout is the evolution of a single black hole. This avoids additional complexities that occur for binaries, most notably apparent horizon finding and MPI. This evolves analytically-computed Kerr-Schild initial data for an isolated black hole, defined by its mass $m$ and its dimensionless spin-vector $\vec{\chi}$. For a black hole with mass $m$ and angular momentum $\vec{J}$, one defines this dimensionless spin-vector as

$$\vec{\chi} = \frac{c \cdot \vec{J}}{G m^2}. \quad (3.7)$$

Black holes must have $0 \leq |\vec{\chi}| \leq 1$. For our test, we choose $\vec{\chi} = (0.2, 0.3, 0.4)$, representing a moderately spinning black hole whose spin axis is not aligned with any of the coordinate axes.

The spectral domains are two nested spherical shells centred on the hole, the first extending radially from $r = 1.62m$ to $r = 6m$ and the second from $r = 6m$ to $r = 12m$. This is sufficient for the simulation to remain stable for at least several thousand timesteps. A full BBH simulation would involve domains besides spherical shells, but spherical shells are the most important, the most individually expensive, and the least friendly to GPU acceleration. The spherical shells in this simulation have 10, 19, and 38 points respectively along their radial, polar, and azimuthal coordinates, for a total
gridsize of 7220 points. We profile throughout from the 5th to the 105th timestep to avoid contamination by simulation setup costs, which are negligible in production runs.

The pie chart of Figure 3.4 illustrates, by percentage, the per-module compute time spent by SpEC during such a simulation at a resolution comparable to that of a binary production run. These modules do the following:

1. Jacobian - contracts the spatial components of a tensor with a Jacobian as part of a coordinate transformation.

2. Deriv - computes the gradient of a tensor (‘Deriv’). This is followed by another Jacobian multiplication (‘Trans’).

3. Filter - maps from physical to spectral space, applies a filter function, and maps back.

4. GHEqns - Computes the right-hand side of the Einstein equations.

5. Apply BCs - extracts the two-dimensional data at the boundaries of the domains, applies the boundary conditions to these, and inserts them back into the three-dimensional volume data.

6. Other - all other operations, each individually negligible.

The rest of this section presents our approach to porting (or justifies leaving unported) each of these.

3.3.2 Memory Management

The physically separate memory of the CPU and GPU mean that data must be kept synchronized between the two devices: accesses to arrays in CPU memory must have a means to ensure that they have not been obsolete by a computation upon their GPU counterparts, and vice versa. Since the actual synchronizations are very expensive, we would like to perform them only when actually necessary. Normally this is handled explicitly by the user, but this approach would in our case require an excessive number of API calls. We have instead developed C++ classes to “lazily” hide memory management from the user. Two arrays are maintained, one on each device, with allocations or copies made only when necessary.

It turns out that GPU memory allocation is extremely expensive. Since SpEC unfortunately makes many allocations and deallocations of memory, the naive approach of allocating and deallocating GPU memory in turn can massively degrade performance.
On the other hand it is clear that repeated allocations will be typically redundant. If an array of a given size is allocated at one timestep, another array of the same size will very likely be allocated at the next timestep as well.

It therefore becomes advantageous to cache rather than deallocate GPU pointers upon array destruction. We handle this by maintaining a map between sizes and cached pointers, declared statically to persist between timesteps. When a GPU allocation of a given size is requested, this map either performs the allocation or retrieves an appropriately sized pointer from the cache. When memory is freed, it is stored in the cache rather than deallocated. True deallocation occurs when the total allocated memory exceeds a certain size, or when explicitly performed by the user.

Usually, our array class DataMesh does not occur alone, but as an element within our container class Tensor. Tensor handles indexing in such a way as to represent a mathematical tensor with a given dimension, rank, and symmetry structure. Quite often, SpEC will perform some operation uniformly on all elements of a Tensor. Due to the kernel launch overhead GPU, it is much more efficient to handle the entire Tensor at once than it is to launch a new kernel for each individual DataMesh.\footnote{For very large gridsizes, launch overhead will be negligible compared to the runtime of individual kernels, and for moderately large ones much of latency can be hidden by the use of concurrent “streamed” kernels. At the gridsizes we are interested in, unfortunately, launch overhead remains a substantial burden.}

SpEC, however, allows the elements of a Tensor to be of arbitrary type, and does not assume they have a uniform memory layout. This is achieved by handling Tensors as arrays of pointers, one per element, which have no relationship to one another in linear memory. On the CPU this is usually not a problem: calling a function once per tensor element is essentially free, as is iterating through the Tensor. But on the GPU calling one kernel per element is quite expensive. A single kernel could process the entire Tensor, but to do so the list of pointer-to-elements would need to be copied to the GPU, and this copy carries again a large amount of overhead.

In practice, the capability of Tensors to store nonuniform objects is rarely used, so we handle this situation with a compromise. Each Tensor stores a list of “GPUPointers”, which is initially empty. When the GPUPointers are explicitly asked for, they are constructed on the host and synchronized with the device. Subsequent accesses to the GPUPointers first check whether the Tensor has been reshaped and synchronize again only if it has. This means that if the same Tensor is accessed multiple times during a timestep (which often happens), the GPUPointers will be synchronized only once.
3.3.3 Jacobian multiplication

SpEC includes two performance-critical modules implementing coordinate changes as contractions with a Jacobian matrix. The first, which we call “DiffJac”, occurs after differentiation, bringing the derivative index into the same coordinate frame as the tensor indices. Indices \((a, b \ldots)\) run over spacetime and \((i, j \ldots)\) over space, while we represent partial differentiation with a comma. Then this operation may be written symbolically as

\[
T_{a\ldots, i} = J_{j}^{i} T_{a\ldots, j},
\]

where \(T_{a\ldots, i}\) is the tensor being transformed and \(J_{j}^{i}\) is the Jacobian of the transformation. The code does not distinguish between contravariant and covariant indices; we do so here only to clarify which indices are being summed over.

The second operation, which we call “SpatialCoordJac”, makes a coordinate change of a tensor’s spatial indices only, by contracting every possible combination of said spatial indices with the Jacobian. First, one contraction per rank is performed over the purely spatial indices. Next, each index is respectively set to its timelike component and one contraction per remaining index is performed. Subsequently, two indices are made timelike, followed by contractions over all unfixed indices, etc. In the case of a rank 2 tensor \(T_{ab}\), for example, this operation may be written

\[
T_{ij} = J_{k}^{i} J_{l}^{j} T_{kl},
\]

\[
T_{0j} = J_{l}^{j} T_{0l},
\]

\[
T_{i0} = J_{k}^{i} T_{k0}.
\]

The purely timelike component encounters a null-op, having been “contracted with no Jacobians”.

In practice, both of these operations are only ever applied to tensors with the following four rank and symmetry structures: \(T_{a}, T_{ab}, T_{aa}, T_{abb}\). The subscripts on the above represent the rank and symmetry structure of the tensor \(T\), with repeated indices indicating a symmetry. Thus \(T_{abb}\) indicates a dimension 4, rank 3 tensor satisfying \(T_{abc} = T_{acb}\). In the case that a symmetry structure other than one of those specified above is encountered, our port falls back on the CPU code.

The actual kernels are fairly simple. They could likely be optimized further, but already perform sufficiently well that Jacobian multiplication is a very small expense on the GPU. For the DiffJac operation of Eq. (3.8), we first copy the pointer addresses of the individual tensor elements into linear GPU arrays. We divide the CUDA grid into two-
dimension thread-blocks. The x-coordinate runs over the spatial grid, the block index of the y-coordinate labels the components of the input tensor, and the thread index of the y-coordinate those of the Jacobian. Each block therefore has local to it all the Jacobian tensor pointers necessary for the contraction, which we load into shared memory to limit register consumption. Each thread performs the contraction in serial.

The SpatialCoordJac operation of Eqs. (3.9)-(3.11) proceeds similarly. Rather than copying individual tensor indices into a GPU array, we bundle them into a struct which is sent to the kernel as a function argument. The pointers are then read into device registers rather than shared memory. We thus need only one-dimensional blocks. Register loads are much faster than shared memory loads, and since the SpatialCoordJac operation involves multiple successive contractions with the same Jacobian this approach improves performance when the register file is sufficiently large and the kernel is bandwidth-bound.

It is nevertheless suboptimal, since the extra register consumption can limit occupancy in practice. Most notably, for \( T_{ab} \) on the M2090 GPU our kernel actually exhausts the available registers, so that the local variables defined in the kernel must be allocated in global memory. This does not happen on the K80 and P100 GPUs, which have larger register files (see Figure 3.5, discussed in detail after the arithmetic intensity models developed below, where the M2090 \( T_{ab} \) benchmark noticeably underperforms). In future versions we will use the shared memory approach for both operations. But the overall speedup is already such that this would not noticeably affect SpEC’s performance (c.f. Figures 3.11 and 3.12).

To estimate the expected performance of these Jacobian multiplications, we need to model their arithmetic intensity \( I \), for use in Equations (3.5) and (3.6) alongside \( w = 8 \) bytes and values of \( B \) and \( P \) read off from Table 3.1. In turn, we need to work out the number of memory operations \( M \) and FLOPs \( F \) each operation entails. The DiffJac operation Eq. (3.8) must read one double per gridpoint per each unique array in \( T_{ab...i} \) and \( J_{ij} \), and then store the results again in \( T_{ab...i} \). We consider the arrays to consist of \( N_x \) elements each, where the coordinate \( x \) runs over physical space. Labelling the number of unique arrays in a tensor \( T \) as \( N^T_e \), and the spatial dimension \( d \), the spatial derivative of \( T \) comprises \( N^T_e d \) unique arrays, the Jacobian comprises \( d^2 \) unique arrays, and the minimum number of memory accesses is \( M = N_x (2 N^T_e + d) d \). The FLOP count is \( F = N_x N^T_e (2d - 1) d \), which can be computed by viewing the operation as one multiplication per spatial gridpoint of an \((N^T_e, d)\) matrix by a \((d, d)\) matrix.

For \( d = 3 \) this yields an arithmetic intensity of \( I = 5 N^T_e / (2 N^T_e + 3) \) instructions per transaction, which notably is independent of the gridsize. For \( T_a, T_{aa}, T_{ab} \) and \( T_{abb} \) respectively we have \( N^T_e = 4, 10, 16 \) and \( 40 \), yielding \( I_a = 1.81, I_{aa} = 2.17, I_{ab} = 2.29, \ldots \)
and $I_{a b b} = 2.41$ (I limits to 2.5 with large $N^T_e$). These values are only somewhat smaller than $I_{e q}$ in Table 3.1 for the three GPUs. We therefore expect both computational and memory throughput to be important performance considerations. However, on the CPU $I_{e q} = 0.41$, indicating that computational performance is important in that case. Thus, we expect CPU-GPU speedups much higher than the simple ratio between the CPU and GPU bandwidths. Specific “ideal” predictions for performance and speedup are given in Figure 3.2.

We now turn to the computation of $I$ for the SpatialCoordJac operation of Eqs. (3.9)-(3.11). In this case $M = N_x (2N^T_e + d^2 - 2)$ memory transactions are necessary; the subtractive factor of 2 accounts for the purely timelike component of $T$ not participating in the operation. For $d = 3$ we then have $M_a = 15 N_x$, $M_{aa} = 30 N_x$, $M_{ab} = 48 N_x$, and $M_{abb} = 120 N_x$.

The FLOP count $F$ is more complex, due to the multiple operations and the fact that the Jacobian is now contracted over possibly-symmetric indices. However, we can immediately see that whatever contractions are necessary will need to be done once per gridpoint. Therefore, $F$ will depend linearly on $N_x$, just as $M$ did, so that $I$ will be independent of $N_x$ for SpatialCoordJac, just as it was for DiffJac.

We nevertheless seek concrete estimates of $F$ for each tensor structure. If symmetries can be neglected, $F$ for a tensor of arbitrary rank $r$ can be computed without much difficulty. The initial spatial contraction, Eq. (3.9), can be viewed as $r$ matrix-multiplications between the $(d,d)$ Jacobian and the $(d,d^{r−1})$ input tensor. This takes $N_x r d^r (2d - 1)$ operations. One index is then (Eq. (3.10)) made timelike, followed by another set of contractions upon an input tensor of rank $r - 1$. There are $r$ different ways to set one index timelike, so this second step is done $r$ times. Eq. (3.10) therefore takes $N_x r (r - 1) d^{r−1} (2d - 1)$ operations. Following through this reasoning for the full series of operations, suppose $W(r,j)$ is the number of unique arrangements of $j$ zeros in a tensor of rank $r$, and $S(r,j)$ is the number of unique spatial components in the tensor so fixed. Then the operation count (for any tensor) is

$$F = N_x (2d - 1) \sum_{j=0}^{j=r-1} W(r,j) S(r,j) (r - j),$$  \hspace{1cm} (3.12)

---

Here and throughout we use the subscripted notation $I_{a b...}/M_{a b...}/F_{a b...}$ to denote the arithmetic intensity/number of memory accesses/number of FLOPs of a particular operation working on a tensor with the subscripted symmetry structure.
which, for a tensor with no symmetries, reduces to

$$F = N_x(2d - 1) \sum_{j=0}^{r-1} \binom{r}{j} d^{r-j}(r-j).$$  \hfill (3.13)$$

This is a rather steep function of $r$. In $d = 3$, Eq. (3.13) it gives $F_a = 15N_x$ and $F_{ab} = 120N_x$.

The combinatorics when symmetric index pairs are allowed are much more involved. Fortunately we are only interested in the simplest two such quantities, $F_{aa}$ and $F_{abb}$. A tensor of rank $r$ with $\sigma$ symmetric index pairs has $S(r, 0) = d^{r-2\sigma}\left(\frac{d+1}{2}\right)^{\sigma}$. The symmetric pair also reduces the number of unique ways to fix indices. For $r = 2, \sigma = 1$ we have 1 term with $S = \left(\frac{d+1}{2}\right)$ and 1 term with $S = d$. In total this gives $F_{aa} = N_x\left(\left(\frac{d+1}{2}\right)2(2d - 1) + d(2d - 1)\right) = 75N_x$.

Finally, for $r = 3, \sigma = 1$, we get one purely spatial arrangement with $S(r, 0) = d\left(\frac{d+1}{2}\right)$. There are two ways to fix only one index. Fixing the non-symmetric index gives a contribution equal to that for $T_{ab}$, while fixing part of the symmetric pair gives the same from $T_{aa}$. Either of the two ways of fixing two indices gives a $T_a$ contribution. In total, we have $F_{abb} = N_x d\left(\frac{d+1}{2}\right)3(2d - 1) + F_{ab} + F_{aa} + 2F_a = 495N_x$.

The arithmetic intensities for SpatialCoordJac are thus $I_a = 1, I_{aa} = 2.5, I_{ab} = 2.5$, and $I_{abb} = 4.125$ (I in general depends on the symmetry structure). Since these numbers are very similar to the ones we obtained for DiffJac, we expect similar performance in both cases. In particular, performance will depend most strongly upon hardware memory bandwidth $B$ on the GPU, and on processing power $P$ on the CPU, and $BW_{\text{eff}}$ will be an appropriate metric of performance.

With these theoretical considerations in mind, we now turn to actual benchmarks. On each of the four architectures listed in Table 3.1, we execute DiffJac and SpatialCoordJac upon tensors of structures $T_a$, $T_{aa}$, $T_{ab}$, and $T_{abb}$. Using the models above and the measured execution time $t$, we then compute $BW_{\text{eff}}$ for each case from Eq. 3.3 5. These $BW_{\text{eff}}$ results are plotted against the spatial gridsize $N_x$ in the top panels of Figure 3.5, whose $x$ axis switches from linear to logarithmic at $N_x = 8000$ in order to compactly display the large-$N_x$ behaviour. Each line on those plots represents a different processor from Table 3.1, indicated by differing colours, or a different tensor structure, indicated by differing linestyles. The bottom panels show the CPU-GPU speedup, i.e. the ratio between the execution time on the indicated GPU and that on the CPU. For both $BW_{\text{eff}}$ and the speedups, higher $y$-axis numbers indicate superior GPU performance.

5 As described in Section 4.5 and illustrated in Figure 3.3, we clean the $BW_{\text{eff}}$ measurements across multiple executions by taking their median.
Figure 3.5: Effective bandwidths (top) and speedups (bottom) for the two Jacobian modules. Line colours and styles respectively indicate different processors and tensor structures. The plots switch from a linear to a logarithmic x-axis scale at a gridsize of 8000.
We now highlight the salient features of Figure 3.5 and interpret them in light of our theoretical expectation from the computation of $I$ and the pragmatics of our implementation. Our predicted arithmetic intensities $I$ were between 1 and 5. Consulting the bottom-left panel of Figure 3.2, we see that at peak performance $BW_{\text{eff}}$ should thus be roughly constant in $I$, and thus independent of both the tensor structure and of whether we are performing DiffJac or SpatialCoordJac.

The expected independence of tensor structure and $N_x$ can, for the GPUs, be seen in Figure 3.5. There, the various linestyles representing differing tensor structures appear for each GPU to converge at large $N_x$. The exception of the $T_{ab\ell}$ kernel running SpatialCoordJac on the M2090, whose performance can be seen from the top right panel of Figure 3.5 to be far beneath the other M2090 curves. In this case, the kernel spills registers into local memory. The CPU curves show a much stronger dependence upon tensor structure than predicted when running SpatialCoordJac, and in both cases show a clear negative dependence upon $N_x$, presumably since large gridsizes overflow the CPU cache. The DiffJac curves, even in the CPU case, also show a sharp decline in performance around a gridsize of $2^6$, the reason for which is not entirely clear.

For both DiffJac and SpatialCoordJac, the P100 outperforms the M2090 by about a factor of 3 at large $N_x$, as can be seen by comparing the rightmost edges of the $BW_{\text{eff}}$ lines in Figure 3.5 for these processors. This is consistent with the expectation from a similar comparison from the bottom left panel of Figure 3.2. These performances are, however, quantitatively each about a further factor of 3 away from Figure 3.2’s prediction, indicating that algorithmic redesign could likely further improve performance. This is especially true for the K80, which actually performs slightly less well than the M2090 despite greatly superior specifications. However, as will be shown in Section 3.4 (specifically Figures 3.11 and 3.12), the speedups we have already obtained make Jacobian multiplications a very small part of the overall black hole simulation runtime, so further effort would have little practical impact.

The GPU performance for SpatialCoordJac at large gridsize is, contrary to the expectation from the above analysis (i.e. from the fact that $I$ is quite similar for both operations), consistently about a factor of 2-5 better than for DiffJac, as can be seen by comparing the large $N_x$ behaviour of identically styled curves for the GPUs on the left and the right panels of Figure 3.5. The relevant kernels are coded somewhat differently: all the necessary pointers-to-tensor elements are first collected into a struct, which is passed to the GPU as an argument to the kernel rather than by a CUDA memory copy. All necessary data are then loaded into registers in a way that interleaves memory accesses with computations. This approach may ultimately entail fewer, or better optimized, memory
accesses, since no explicit pointer indirections are coded in. The staggered instructions may also improve latency, since less time need be spent waiting for data.

### 3.3.4 Spectral Operations: Differentiation and Filtering

SpEC is named for its use of the pseudospectral collocation method [60, 61, 75, 76, 122]. In this section we describe our porting strategy for two operations, differentiation and filtering, which make explicit use of these methods. Let us begin by introducing spectral methods in the simplest case of 1D PDEs and scalar variables. Spectral methods represent the solution \( u(x) \) as a series expansion in basis functions \( T_k(x) \):

\[
    u(x) = \sum_{k=0}^{N-1} \tilde{u}_k T_k(x).
\]  
(3.14)

The \( \tilde{u}_k \) above are called spectral coefficients. The approximation arises because \( N \) is finite.

Furthermore, there is a set of collocation points

\[
    x_i, i = 0, \ldots, N - 1.
\]  
(3.15)

The function values at the collocation points, \( u_i \equiv u(x_i) \), can be computed by a matrix multiplication:

\[
    u_i = \tilde{M}_i^k \tilde{u}_k,
\]  
(3.16)

where \( \tilde{M}_i^k = T_k(x_i) \). For suitable choice of collocation points, the inverse is also a linear transformation:

\[
    \tilde{u}_k = M_k^i u_i.
\]  
(3.17)

We shall refer to Eq. (3.16) as “SpecToPhys” and to Eq. (3.17) as “PhysToSpec”.

For a Fourier series and Chebyshev polynomials, the transforms Eq. (3.16) and Eq. (3.17) can be evaluated respectively with a fast-Fourier-transform or fast-cosine-transform (we hereafter loosely use the term “FFT” to refer to both possibilities), with \( \mathcal{O}(N \log N) \) complexity scaling rather than the naively-expected \( \mathcal{O}(N^2) \) for matrix-vector multiplication.

The advantage of spectral methods is that many operations of interest, most notably including differentiation, can also be performed with an FFT, yielding \( \mathcal{O}(N \log N) \) complexity scaling overall. Since, for example, the basis functions form a complete set, their
derivatives are linear combinations of basis functions

\[ T'_k(x) = \sum_{k}^{N-1} \bar{D}_k T_l(x) \]  

(3.18)

which we can exploit to find the differentiation matrix $\bar{D}_k$ analytically. Multiplying this matrix by the spectral coefficients $\tilde{u}_k$ - which, again for a Fourier series and Chebyshev polynomials, can be done with a $O(N \log N)$ FFT - yields $\tilde{u}'_k$, the spectral coefficients of the solution’s derivative. The real space derivative $u'_i$ can then be obtained using Eq. (3.16), with FFT-like scaling overall.

Differentiation accounts for around 20% of SpEC’s total runtime during the SingleBH test (c.f. Figure 3.4). A second operation called the spectral filter consumes about an additional 30%, and is very similar in form. Here, we perform the same spectral transformations as in (3.17) and (3.16), but the matrix in (3.18) is designed to apply some filtering transformation to the spectral coefficients rather than to compute derivatives. For example, the Heaviside filter zeros out all Fourier modes with a frequency above a certain value.

In practice, SpEC maintains for each one of these steps a complicated battery of C++ classes that are appropriate for different choices of simulation domain, spectral basis function, and low-level implementation details. Producing hand-written CUDA equivalents of all possible execution pathways would take, to say the least, significant effort. In particular code maintenance would become unmanageable. To avoid this we capitalize on three features of the spectral operations. First, they are all linear maps. Second, a given process will encounter only manageably few unique instances of each. Third, in a practical simulation each unique instance will be encountered by a process very many times.

These properties in concert make feasible a general strategy based on tracing out an explicit matrix representation for each function. Specifically, considering for example the differentiator, we express the entire transformation as an explicit multiplication with a single matrix $D^i_j = (\bar{M}DM)^i_j$. We then have

\[ u'_j = D^i_j u_i \]  

(3.19)

which is mathematically equivalent to the differentiation operation, however the latter is implemented. We can exploit this fact to trace out an explicit matrix representation of $D^i_j$. Specifically, we set $u_i = \delta_{ik}$ for some $k$, where $\delta_{ik}$ is the Kronecker delta, and then pass this input through the actual extant CPU code. The result is the $k$th column of the
matrix $D_{ij}$. By repeating this procedure for all $k$'s, we trace out the entire matrix in $N_x$ function calls.

Having traced out $D_{ij}$, we maintain an associative array (which we call a dictionary) between it and whatever function input specifies a mathematically-new transform. For the differentiator, this is the gridsize and derivative index, while for the spectral filter, it is the gridsize, the particular filter function to be applied, and in some cases the tensor structure of the input. The extra function calls needed to build the matrix are in practice very few compared to the full number that will be made over the SpEC runtime, so the extra expense can be ignored.

In general, implementing the spectral operations by explicit matrix multiplications such as Eq. (3.19) will result in worse asymptotic complexity than is achieved by the spectral CPU code, whose expense is dominated by either an FFT or a closely related algorithm. Nevertheless, this approach can be advantageous, especially when viewed as a CPU-to-GPU porting strategy. First, instead of needing to port, optimize, and maintain a parallel GPU code for each of the very many possible spectral transformations, only one or a small number are necessary. Second, the operation counts at low-$N$ can be such that matrix multiplications actually outperform fast transforms at practical gridsizes. Third, FFTs are, in general, much more difficult to parallelize than matrix multiplications, leading to much lower FLOP/s for the former: for example NVIDIA reports large-$N$ double precision operation rates of around 150 GFLOP/s for their cuFFT library running on a K40 GPU [227], compared to near-peak performance in the TFLOP/s regime for matrix multiplication [225] (of course, the FFT involves many fewer operations). Finally, matrix multiplication can be performed by the (cu)BLAS function `dgemm`, which is possibly the most heavily optimized function in existence.

Let us now consider the realistic case of 3D grids and tensorial solution variables. Usually, the independent tensor components are decoupled, and so generalizing to tensors with $N_e$ independent components simply involves a factor of $N_e$ extra function calls. But the higher spatial dimensions are qualitatively important, since they change the shapes and characters of the matrix multiplications (or fast transforms).

Let us now consider our port of the differentiator as it works in practice. We start with a function $u(x, y, z)$ available at physical collocation points $u_{ijk} = u(x_i, x_j, x_k)$, and denote by $N_x$, $N_y$, and $N_z$ the physical gridsizes in the subscripted dimension $^{6}$. The full domain topology is an outer product of so-called “irreducible topologies” which cannot be themselves expressed as outer products, and each irreducible topology will be associated with its own set of spectral basis functions. These basis functions will depend on each

$^{6}$While our discussion centres upon $d = 3$, generalization to other dimensions will be obvious.
physical coordinate on their associated domain. For example, on an $I^1 \otimes I^1 \otimes I^1$ domain, where the irreducible topology $I^1$ is that of a closed line segment, we have three sets of spectral basis functions, and each set depends on only one physical coordinate; we thus call the basis functions 1D. In this case we write

$$u(x, y, z) = \sum_{i,j,k} \tilde{u}_{ijk} T_i(x) T_j(y) T_k(z). \quad (3.20)$$

In particular, we can obtain e.g. the $\tilde{u}_i$ coefficients, which are all we need to compute the $x$-derivative, without performing the other two sums:

$$\tilde{u}_{akj} = M_{\alpha}^i u_{ikj}. \quad (3.21)$$

The derivatives are again linear combinations of basis functions, we again finish by mapping back to the collocation points, and we again can trace out an explicit matrix representation of the entire operation by feeding delta function input through the CPU code. Denoting this matrix representation by the capital letter corresponding to the physical coordinate upon which it operates, we have

$$u_{abc,d} = \delta_{d1} X_a^i u_{ibe} + \delta_{d2} Y_b^j u_{ajc} + \delta_{d3} Z_c^k u_{abk}. \quad (3.22)$$

In some cases SpEC works upon domains composed of irreducible topologies which are not 1D in the above sense - that is, their associated spectral basis functions depend on more than one physical coordinate. The most notable example is spherical shells, with topology $I^1 \otimes S^2$. The $I^1$ irreducible topology, representing the radial direction $r$, admits a spectral basis of 1D Chebyshev polynomials that depend only on $r$, but the spherical harmonics $Y_{lm}$ depend on both angular coordinates, $\theta$ and $\phi$. SpEC furthermore in this case uses a compressed, but slightly redundant, spectral representation, so that $\tilde{M}_i^k$ is not simply the inverse of $M_k^i$. The matrix product $\tilde{M}_i^k M_k^j$ projects into a subspace of $\{u_i\}$.

In this case, we have physical variables $u(r, \theta, \phi)$ available at physical collocation points $u_{klm} = u(r_k, \theta_l, \phi_m)$. The physical gridsizes in the $r$, $\theta$, and $\phi$ directions will be denoted $N_r$, $N_\theta$, and $N_\phi$, while the number of $l$ and $m$ $Y_{lm}$ modes maintained in the spectral representation will be called $N_l$ and $N_m$ (the number of Chebyshev coefficients is just $N_r$). The spectral coefficients are written

$$u(r, \theta, \phi) = \sum_{klm} \tilde{u}_{klm} Y_{lm}(\theta, \phi) \quad (3.23)$$
so that computation of either angular derivative requires the entire double sum over both \( l \) and \( m \). We end up with

\[
  u_{abc,d} = \delta_{d1} R^i_{a} u_{ibc} + \delta_{d2} \Theta_{bc}^{jk} u_{ajk} + \delta_{d3} \Phi_{bc}^{jk} u_{ajk}.
\]  

(3.24)

There is now the practical business of expressing these operations as sequences of BLAS calls\(^7\). \texttt{SpEC} stores the collocation data \( u_{ijk} \) as physically contiguous arrays, so that we are free to join together adjacent indices. We may thus view the data equivalently as a matrix \( u_{i,jk} \) (for the first transform), as a matrix \( u_{i;j,k} \) (for the last, or the last two in the \( I_1 \otimes S_2 \) case), or as a set of \( k \) submatrices \( u_{i;j} \) (for the middle transform), each one of which is multiplied by the appropriate transformation matrix. The colon notation above indicates vectorization into a single index; i.e. \( u_{x,y,z:k} \) is a matrix of size \( (N_x, N_y N_z) \).

Since these are just sequences of matrix multiplies, we can easily estimate FLOP and memory transaction counts for each. For \( I_1 \otimes I_1 \otimes I_1 \), we first shape the input as an \( (N_x, N_y N_z) \) matrix and multiply with the \( (N_x, N_x) \) x-transform matrix. Next we shape the input into \( N_z \) \( (N_x, N_y) \) submatrices and multiply each with the \( (N_y, N_y) \) y-transform matrix. Finally, we shape it into an \( (N_x N_y, N_z) \) matrix and multiply that with the \( (N_z, N_z) \) z-transform matrix. These “reshapings” are just parameter choices to BLAS, and involve no actual copies. We repeat this procedure once for each of the \( N_e \) independent components of the input tensor. The operation count for any particular coordinate \( x_d \) with size \( N_d \) has the functional form

\[
  F = N_e N_x N_y N_z (2N_1 - 1).  
\]  

(3.25)

The x and z transforms involve

\[
  M = N_e (N_d^2 + 2N_x N_y N_z)  
\]  

(3.26)

memory operations. As formulated above, however, the y-transform matrix must be read in by the device \( N_z \) times, with each read acting upon a fraction \( 1/N_z \) of the entire volume data. We thus have

\[
  M = N_e N_y N_z (N_y + 2N_x).  
\]  

(3.27)

These memory access estimates somewhat exceed what is strictly required. The transform

---

\(^7\)BLAS accepts ‘transpose’ parameters which determine whether the input matrices are to be read in standard (’N’) or transposed (’T’) format. \texttt{SpEC} stores physical data in row-major format, but BLAS assumes column-major, so the input is implicitly transposed anyway. By chance, this naturally leads to the choice ’N’,’N’ for the first basis function and ’T’,’N’ otherwise, which are the two most favourable cases in terms of performance.
matrices, for example, are the same for each tensor component, and a kernel could load them from global memory only once for the entire tensor. Optimizing such a kernel to outperform cuBLAS even given the extra accesses would, however, be difficult, especially since matrix multiplication is compute-bound. If SpEC stored entire tensors as contiguous arrays this could be achieved using the cuBLAS function `cublasdgemmStridedBatched`. Since this is not in fact the case, we use the above accounting.

In total, we have

\[ M = N_e(6N_xN_yN_z + N_x^2 + N_y^2 + N_z^2) = N_e(7N^3 + 2N^2) \]

\[ F = N_eN_xN_yN_z[2(N_x + N_y + N_z) - 3] = N_e(6N^4 - 3N^3). \]

where the rightmost equalities assume \( N_x = N_y = N_z \equiv N \), in which case the arithmetic intensity is \( I = N^2(2N-1)/(7N^2+1) \sim 2/7N \) (note again that \( N \) here is the linear gridsize). Comparing with \( I_{eq} \) from Table 3.1, we see this operation will be compute-bound at any realistic input size. For \( I_1 \otimes S_2 \), the transform matrix shapes are \((N_r, N_r)\) for the radial transform and \((N_\theta N_\phi, N_\theta N_\phi)\) for both of the angular ones. The input reshapings are \((N_r, N_\theta N_\phi)\) and \((N_\theta N_\phi, N_r)\). Due to the dependence of the spherical harmonic basis functions on both angular coordinates, the transforms are also over both, even if we only seek e.g. the \( N_\theta \) derivative. For the same reason, we do not break into \( N_\phi \) submatrices for the \( N_\theta \) transform, as we did for \( N_y \) in \( I_1 \otimes I_1 \otimes I_1 \).

Noting that \( N_i = N_\theta N_\phi \) for the angular transforms, \( F \) and \( M \) have the same forms as in (3.25) and (3.26). In total, we have

\[ M = N_e(N_r^2 + 2N_\theta^2N_\phi^2 + 6N_rN_\theta N_\phi) \]

\[ F = N_eN_rN_\theta N_\phi[2N_r + 4N_\theta N_\phi - 3]. \]

In practical simulations, the resolutions \( N_r, N_\theta, \) and \( N_\phi \) can differ widely from one another. For our benchmarks we therefore distribute points by two prescriptions. The “SingleBH” benchmarks are on spherical shells that mirror those found in SpEC’s isolated black hole evolutions. This is the gridpoint distribution that SpEC would use in, for example, studies of perturbations to such isolated black holes. Resolution is controlled by a resolution parameter \( k = 0, 1, \ldots, 10 \), in terms of which we have \( N_r = 9 + 4k, N_\theta = 6 + 2k, N_\phi = 4k + 12 \), and thus \( N_\theta N_\phi = 8k^2 + 48k + 72 \). The “BBH” benchmarks use roughly the same point distribution as used initially for the spherical shells closest to the apparent horizons of black hole binaries in an actual BBH simulation.\(^8\) Note this

\(^8\)SpEC employs adaptive mesh refinement during a run, so the actual point distribution during a
Figure 3.6: Top: Arithmetic intensity $I$ plotted as a function of gridsize $N$ for the matrix multiply differentiator operating on $I_1 \otimes I_1 \otimes I_1$, and upon $I_1 \otimes S_2$ with the SingleBH and BBH gridpoint distributions. For $I_1 \otimes I_1 \otimes I_1$ these estimates also apply to the spectral filter. Bottom: theoretical zero-latency processing rate $P_{\text{eff, opt}}$ for each of the benchmarked devices as a function of gridsize. Note that the CPU uses a more efficient (at large gridsize) algorithm, so these lines do not bound its performance.

The radial resolution is comparatively much lower in this case. Specifically we have $N_r = 4 + k$, $N_\theta = 7 + 2k$, $N_\phi = 4k + 14$, $k = 1, 2, \ldots, 16$, and thus $N_\theta N_\phi = 8k^2 + 56k + 98$. For SingleBH, we have

$$M = N_e(128k^4 + 1728k^3 + 8512k^2 + 18216k + 14337)$$  
$$\sim 128N_e k^4$$ \hspace{1cm} (3.32)  

$$F = N_e(1024k^5 + 14848k^4 + 85536k^3 + 244728k^2 + 347760k + 196344)$$  
$$\sim 1024N_e k^5.$$ \hspace{1cm} (3.34)

The arithmetic intensity $I = 13.7$ for $k = 0$ and grows approximately linearly thereafter.
For the BBH benchmarks we have

\[ M = N_e(128k^4 + 1840k^3 + 9937k^2 + 23892k + 21576) \]  
\[ \sim 128N_e k^4 \]  
\[ F = N_e(256k^5 + 4624k^4 + 33368k^3 + 120252k^2 + 216426k + 155624) \]  
\[ \sim 256N_e k^5. \]

This arithmetic intensity \( I \) starts at 7.2. The operations will clearly be compute-bound in all cases. Of course, equal \( k \) implies different total gridsize between cases, so it is difficult to estimate performance at equal gridsize from the above. To do that we refer to Figure 3.6, where \( M, F, \) and \( I \) are shown for each of the three grids. With \( I \) in hand as a function of gridsize, we can refer once more to Figure 3.2 to predict ideal effective processing rates \( P_{\text{eff, opt}} \) from Eq. (3.6), which we plot against gridsize in the bottom panels of Figure 3.6.

We now turn our attention to spectral filtering, focussing initially upon \( I1 \otimes I1 \otimes I1 \) topologies. Spectral filtering of 1D basis functions is similar to differentiation, the only difference being the specific form of the transformation matrix. In Figure 3.7 we thus show the performance of both the differentiator and the spectral filter operating on an \( I1 \otimes I1 \otimes I1 \) topology. Comparing the performance of the spectral filter with that of the differentiator, we see near-identical behaviour on the CPU. On the GPU we get qualitatively similar but somewhat worse performance from the spectral filter. This is due to the extra cost in the latter case of looking up the cached transform matrices. Since the differentiator always implements the same transformation, we can store the relevant matrices as private members of a differentiator C++ class. For the spectral filter, there are very many possible transformations, which necessitates a more complicated caching strategy. While the performance difference is likely unimportant in practice, the lookup could probably be substantially optimized if necessary.

The CPU curves in Figure 3.7 are computed using the same operation count model as we use in the GPU case, which is \( O(N^4) \) in the linear gridsize \( N \). However, the CPU in practice uses an FFT on the transformed basis function, and so its true scaling is, for favourable collocation point choices, \( O(N^3 \log N) \). Because our model underestimates the true CPU FLOP count the CPU performance curves on Figure 3.7 can in principle exceed the CPU’s theoretical performance (c.f. Figure 3.6), although in this case they do not. The scaling coefficients at lower gridsizes are better for matrix multiply, which is why the latter algorithm can be favourable, especially given superior hardware. Unfavourable collocation point choices can furthermore affect the true CPU FLOP count by about an
Figure 3.7: Effective processing power $P_{\text{eff}}$ and speedups vs. one CPU core for the matrix multiply differentiator (left) and spectral filter (right) acting on an $I_1 \otimes I_1 \otimes I_1$ topology. Input tensor structures differ by linestyle, while the devices of Table 3.1 differ by colour. The CPU algorithm exhibits sharply gridsize-dependent performance, and we compute speedups only at peaks, marked with black circles. In the top (bottom) panels, we use the batched (streamed) API, which in this case performs better (worse).
order of magnitude, causing the jagged behaviour of the CPU curves in Figure 3.7. When computing the speedup, we use only the “peak” points (chosen by eye), and have plotted a linear interpolation between these.

The performance of the CPU algorithm is roughly independent of the number of independent tensor components $N_e$. For the GPU algorithm we can get some dependence upon the latter. The individual matrix multiplication sizes are on the order of the linear gridsize, between around 10 and 40. Neither cuBLAS nor the GPU itself are very well optimized for such small matrix multiplications, which cannot individually utilize all the streaming multiprocessors of the device. This likely accounts for the underperformance of the GPUs compared to their theoretical processing powers.

We thus use one of two concurrency strategies that allow multiple small kernels to exhibit some parallelism. The first strategy, called “streamed”, attempts to run the kernels concurrently using CUDA streams. These are a CUDA API feature that allow kernels to be run asynchronously with the CPU and with one another. This approach cannot achieve concurrent execution for very small kernels for which the kernel launch overhead of about 20 $\mu$s is an important expense, since only one kernel can be prepared for launch at one time. Also, since the individual kernels have no knowledge of one another, cuBLAS must tune them as if they were to run synchronously, which may result in suboptimal tuning overall.

The second strategy, called “batched”, runs each separate matrix transformation as a single call to the API function cublasDgemmBatched. This function performs an identical matrix multiplication on a series of matrices, given to the API as an array of pointers. Using it incurs some extra overhead, since this array must be first copied to the GPU. The batched API can in many cases give superior performance to streamed multiplications. Generically, it will be the better choice for numerous multiplications on small kernels. In that case the batched API can save on launch overhead, and may also make superior tuning choices since it is aware of the full operation.

Sometimes, however, the streamed strategy is favourable. It is not easy to predict which will be which except by experiment. We have, for example, performed benchmarks which show that for some matrix shapes the batched strategy is a factor of 2-5 faster even when only a single (small) matrix is being operated upon. In other cases, we have found that the batched API is modestly superior on some cards, but that streamed calls are almost an order of magnitude better on newer ones, presumably because of new GPU

---

9There is another API function, cublasDgemmStridedBatched, which avoids this overhead by accepting a single pointer for each matrix along with a stride that determines where in GPU memory each new matrix begins. We are unable to use this function since our Tensor elements are not respectively contiguous.
Figure 3.8: Effective processing power $P_{\text{eff}}$ and speedups vs. one CPU core for the matrix multiply differentiator acting on a $I_1 \otimes S_2$ topology using the ‘SingleBH’ gridpoint distribution, using the streamed (left) and batched (right) concurrency strategy. In terms of the resolution parameter $k$, ‘SingleBH’ has $N_r = 9 + 4k$, $N_\theta = 6 + 2k$, and $N_\phi = 2N_\theta$. Different linestyles indicate differing tensor structures as indicated in the legend, with a colour fill between $T_{\text{abb}}$ and $T_a$ (performance of the intermediate structures is usually, but not always, bounded by these).

features being exploited on newer cards. Because of this, we have experimented with both strategies in all our benchmarks.

On $I_1 \otimes I_1 \otimes I_1$, the batched strategy consistently gives an improvement of about an order of magnitude, larger tensors giving a greater advantage. This topology involves very many individually tiny matrix multiplications ($N_e(2+N)$ in total), so this is perhaps to be expected. Especially when using the batched strategy, we get very impressive speedups overall, of between 10 and 100X. This is despite the observed performance being about an order of magnitude beneath our theoretical prediction. It must be stressed that our CPU benchmarks use only 1 CPU core, which has a fairly modest clock frequency of 2.0 GHz. While realistic for SpEC this would in most circumstances be a very unusual comparison. 6 CPU cores running at 3-4GHz might be more typical of modern hardware, which would give about an order of magnitude speedup assuming linear scaling with parallelism (which SpEC cannot achieve).
We now turn to the results on spherical shells $\mathcal{I}1 \otimes S2$, where a more complex picture will emerge than for $\mathcal{I}1 \otimes \mathcal{I}1 \otimes \mathcal{I}1$. We first discuss the differentiator, results from which are summarized in Figures 3.8-3.9. Especially for larger gridsizes and especially on the P100 our GPU performance is quite comparable to the predicted peak performance shown in the lower panels of Figure 3.6. The CPU performance, on the other hand, exceeds both this prediction, and the CPU’s theoretical processing power. The expense of the transform is dominated by the angular sector in both cases. On the CPU, the $\phi$ transform is done with an FFT, and the $\theta$ by a matrix multiply, yielding $N_r (N_\theta N_\phi)^3 \log (N_\theta N_\phi)$ scaling, compared to the $N_r (N_\theta N_\phi)^4$ scaling of our model. This gives a ratio $N_\theta N_\phi / \log (N_\theta N_\phi)$, which is a larger factor than for $\mathcal{I}1 \otimes \mathcal{I}1 \otimes \mathcal{I}1$, since $N_\theta N_\phi$ is larger. This is particularly true for the BBH case, explaining the improved CPU performance of BBH vs. SingleBH.

For the SingleBH grid we achieve an appreciable speedup of between 5 and 30X throughout. Performance is consistently better for the streamed strategy in this case, perhaps because the individual angular multiplications are now large enough that cuBLAS
can make effective tuning choices. For BBH, where the spectral algorithm gives the largest advantage, the GPU advantage is more modest and the CPU actually exceeds the M2090 performance in some cases. This is unfortunately the more realistic gridsize choice for production simulations.

The batched vs. streamed picture is here much less clear than it was for $I_1 \otimes I_1 \otimes I_1$. On the P100 the streamed strategy is greatly advantageous, whereas batched is modestly superior on the other cards. GPU performance seems in most cases to scale up to some kind of threshold, after which point there is a sharp dip and a new slow scaling upwards (for example at around gridsize 30000 for the BBH batched K80 and M090 runs, or 15000 on SingleBH). This may be due to cuBLAS switching here to a new kernel optimized for multiplications with a large shared dimension. In that case, kernels need to read much more data than they will end up writing to global memory, which limits parallelism.

We now turn to spectral filtering on $I_1 \otimes S^2$ shells. The radial filter, along $I_1$, is handled in the same way as the $x$-dimension in $I_1 \otimes I_1 \otimes I_1$. In our benchmarks, as in production runs, we do not include a filter along the $r$-axis.

For spectral filtering of 2D basis functions, the filtering transformation will normally couple together elements from different components of a tensor. This necessitates a different approach, especially since the relevant coupling will be very sparse. For filtering on $I_1 \otimes S^2$ we therefore break the operation into three steps. During “PhysToSpec” (“SpecToPhys”), each of the input tensor $U_{ijk...}$’s $N_e$ independent components are separately transformed as in Eqs. (3.17) and (3.16). In physical space, each tensor component is viewed as a $((N_r, (N_\theta N_\phi)), N_s)$ matrix, and the spectral transform matrix has dimensions $((N_\theta N_\phi), N_s)$, where the spectral dimension $N_s$ (very roughly $N_\theta N_\phi/2$) is the number of coefficients of the spectral representation. Thus,

$$F = N_e N_r N_s (2 N_\theta N_\phi - 1),$$

$$M = N_e (N_r N_\theta N_\phi + N_r N_s + N_\theta N_\phi N_s).$$

The “SpecToPhys” transformation simply swaps $N_\theta N_\phi$ with $N_s$ in the above.

We choose BLAS parameters such that the PhysToSpec transform maps the different tensor components into a single contiguous $N_e N_r N_s$ array, which we view as an $(N_e N_s, N_r)$ matrix. The filter is then implemented by multiplying with an $(N_e N_s, N_e N_s)$ transform matrix which couples the $N_e$ distinct tensor components. Typically, only about 10% or fewer of the entries in this matrix are nonzero, and so it becomes worthwhile to store it in a sparse format. We use the CSR format [228] because it is fairly simple and well-supported by the NVIDIA sparse algebra package cuSPARSE.
The complexity of the filtering step depends somewhat sensitively upon the actual structure of the sparse filtering matrix and upon the details of the matrix multiplication algorithm. The sparsity of the filtering matrix will in turn depend on what filter is being applied, so we profile using two different such functions, which we call Heaviside (a Heaviside filter) and ExpCheb (an exponential Chebyshev function). We use the cuSPARSE algorithm dcsrm2. The cuSPARSE documentation [228] describes this algorithm as memory bound, with an approximate complexity of $N_e N_s \left[ s N_e N_s (N_e N_s + 1) + 2 N_r \right]$. Here the sparsity factor $s = Nnz/(N_e^2 N_s^2)$, while $Nnz$ is the number of nonzero entries in the sparse matrix.

The benchmarks in this case are illustrated by Figure 3.10. GPU performance is dominated by the (dense) spectral transform multiplications, and so the results are comparable to, but worse than, those of the differentiator acting on $I_1 \otimes S2$, shown in Figures 3.8-3.9. The worse performance is due to the larger number of operations, the extra time needed for matrix lookup, and the more asymmetrical matrix dimensions. Speedups, however, are in many cases higher for the spectral filter, because in that case the CPU performance is much more sensitive to the number of independent components of the input tensor. Curiously, the CPU processes $T_{ab}$ considerably more efficiently than $T_{aa}$, even though the latter has fewer components. As for the differentiator, the streamed concurrency strategy gives somewhat better results overall for SingleBH. For BBH, the batched API is marginally superior except on the P100, where the streamed strategy outperforms by about a factor of 2 (c.f. the hollow lines in the lower panels of Figures 3.11 and 3.12).

### 3.3.5 DataMesh Operations, Apply BCs, GHEqns

Apart from the individually-significant operations described above, SpEC contains numerous operations on our array class, DataMesh, which are distributed too widely throughout the code to port individually. To deal with these we use our automatic porting system, TLoops. While TLoops is the subject of Chapter 4, we briefly describe it here to keep this chapter self-contained.

TLoops furnishes a set of C++ classes to represent arrays, tensors, indices over tensors, and operations between tensors. These classes are based on templates, which recursively iterate at compile time to form unique types for each tensor manipulation written in the SpEC code. After compilation a separate executable can be used to generate valid CUDA code for each unique operation. This can then be linked back to a separate checkout of SpEC. In this way all manipulations of DataMesh throughout the code can be ported at
Figure 3.10: Effective processing power $P_{\text{eff}}$ and speedups vs. one CPU core for the matrix multiply spectral filter acting on an $I_1 \otimes S_2$ topology for ‘SingleBH’ (left) and ‘BBH’ (right) gridpoint distributions. We study two filter functions, Heaviside (top) and ExpCheb (bottom). Speedups are shown only for $T_{\text{abb}}$ and $T_a$. Line colours (and fill) distinguish between processors, and line styles between tensor structures. Here we only profile the streamed API, which generally performs better.
If no changes are made to the code at all, \texttt{DataMesh} operations ported automatically with \texttt{TLoops} will typically show only a modest speedup or even a slowdown at lower grid-sizes, due to large amounts of launch overhead incurred by loops over kernel launches\textsuperscript{10}. Even in the case of a small slowdown, however, the automatic porting yields a net benefit since it avoids numerous CPU-GPU synchronizations that would otherwise occur around the explicitly ported modules.

Much of the slowdown comes from the operations collected as ApplyBCs. These functions operate mostly on angular slices at the boundaries of the domains, which have about an order of magnitude fewer points than do the full three-dimensional volume arrays: a shell with \((N_r, N_\theta, N_\phi)\) gridpoints has boundaries with only \((N_\theta, N_\phi)\) gridpoints, and \(N_r\) is typically around 10. Such operations can be a considerable bottleneck for two reasons. First, the code that extracts and inserts these two-dimensional slices out of and into the volume involves an unavoidably strided data access that is very inefficient to port on the GPU. It is nevertheless best to do so in order to avoid extra synchronizations. Second, the ApplyBCs operations are simply very small and very numerous. Launch overhead impairs their performance severely.

We deal with this by leaving boundary data on the CPU throughout. \texttt{TLoops} expressions check the dimension of the relevant \texttt{DataMesh}es, and execute on the GPU only for dimension 3 or higher. The \texttt{DataMesh} copy constructor also transfers data on the host (rather than the device) for dimension 1 or 2, unless the data is on the device already (for dimension 3 we always synchronize with the GPU and copy there). \texttt{TLoops} still somewhat impairs the performance of the ApplyBCs operations since the boundary arrays must be copied to the GPU whenever they are to be extracted from or inserted to the volume. But the net effect is a speedup.

Launch overhead can be mitigated even further by operating on whole tensors with a single kernel. Code must be modified to do this, so it is not practical to do throughout, but it does provide a very simple and convenient porting strategy for complicated operations. We have used this strategy to port the GHEqns, which solve the Einstein equations. This allows for about a 10X speedup at realistic gridsize without writing any explicit CUDA code (c.f. Section 4.4).

\textsuperscript{10}Concurrent kernel execution using CUDA streams does not help in the case where launch overhead is more expensive than the kernel itself.
3.4 Benchmarks of Overall Code

Figures 3.11 and 3.12, finally, summarize our entire GPU porting results. These figures show benchmarks from runs of SpeC upon isolated black hole test cases. Shown are two gridsizes, SingleBH and BBH, identical to the eponymous gridsizes used in the performance analysis of the differentiation and the spectral filter in 3.3.4. Compared to the SingleBH tests, the BBH tests have a relatively larger angular resolution at constant gridsize. We ran each benchmark five times for 110 timesteps, and collected results between timesteps 5 and 105. As in the previous benchmarks, the plotted results are the median times over the five runs.

These isolated black hole runs evolve a stationary, isolated black hole with a spin of 0.5 using the generalized harmonic equations. Surrounding the black hole are two $I^1 \otimes S^2$ subdomains with identical resolutions. Apart from being numerically simpler, the isolated case differs from a full binary black hole simulation in several ways. A binary black hole simulation would have a much more diverse set of domains, and would involve AMR, which we have not considered here. Binary evolutions also involve interpolations and computations on the apparent horizons of the black holes, which can be importantly expensive. Finally, binary evolutions use MPI, which assigns each individual domain to a different CPU core. In our full port, each would instead be assigned to a separate GPU. However, in these tests, we do our computations on the two respective spheres in serial.

The GPU performance is especially strong in the (less realistic) SingleBH gridpoint distribution, which has more points in the radial direction at a given gridsize. Generally we see comparable performance for both distributions as expected from our individual benchmarks in the per-module speedups, although the differentiator on the BBH distribution performs somewhat worse than expected. The GHEqns, which are ported automatically using TLoops with benchmarks presented in Chapter 4, show particularly strong performance - a 100X speedup on the P100 - especially given that no algorithmic redesign was required here.

Much of the speedup is limited by the ApplyBCs and the Other operations. The latter are mostly a large mass of tensor manipulations which have not been explicitly replaced with TLoops expressions. We expect a further speedup of these modules by 2-15X could be achieved by doing so, due to both savings in launch overhead and extra
parallelism over tensor structure.

The ApplyBCs operations present a more serious challenge. These operations are on arrays of dimension 2, which we have purposely left on the CPU throughout. The performance of these operations is thus limited in principle by the CPU performance\textsuperscript{11}. It will usually be somewhat worse than this, because these low-dimension arrays arise as slices of higher-dimensional data living on the CPU, and extracting these slices requires a GPU-CPU memory transfer. The arrays in question are so small that kernel launch overhead is the dominant expense if they are kept on the GPU throughout, to a sufficient extent that the memory copies are still cheaper overall. However, porting to TLoops may mitigate this.

Parallelising a particular domain across multiple GPUs would not be worthwhile, as our essential problems throughout have been code complexity and the small size of our operations compared to those for which the GPU is optimized. However, multiple GPUs can still be leveraged by assigning one domain to each, which should give roughly linear scaling.

\section{3.5 Conclusions}

We have performed a CPU to GPU port of the portions of the numerical relativity code \texttt{SpEC} relevant to single black hole simulations. Our combined strategy of explicit porting for the Jacobian multiplications, semi-automatic porting for spectral operations that can be written as matrix transformations, and completely automatic porting for the many scattered tensor operations throughout the code gives comparable to peak performance for many of these modules, module-to-module speedups compared to one CPU core ranging from 10 to 100X, and overall speedups of between 2-10X. Due to its reliance on prepackaged libraries, our port also generally shows improved performance when run on newer hardware without requiring extra tuning.

The next step will be to completely port the code so that the accelerated version can be used for production runs. To do this, modules such as the apparent horizon finder which are not present for isolated black hole tests will need to be ported. The port will also need to be made aware of MPI and able to scale to multiple GPUs. Many domains besides spheres will furthermore become important. We expect larger overall speedups on such domains, due to the strong performance on $I_1 \otimes I_1 \otimes I_1$ we have observed in benchmarks. For spheres, it may be worthwhile to directly port the spectral code in some

\textsuperscript{11}We get a slight speedup at times on the K80 and P100 operations because these GPUs are driven by POWER8 CPUs with a higher clock frequency than we used for our CPU benchmarks
Figure 3.11: SpEC’s overall performance broken down by module using the SingleBH gridpoint distribution. On the top and left, we show “stack” plots for each benchmarked device. The height of each slice is the runtime of its module, and the total height is the overall runtime. Dotted vertical lines mark the profiled resolutions. On the CPU stack (top), we overlay the GPU runtimes as black lines. The grey stack (‘Other’) represents all modules not otherwise included. On the right, we show the speedup of each module, with identical colouring as on the left, and the total speedup in black.
Figure 3.12: SpEC’s overall performance broken down by module using the BBH gridpoint distribution. The plots are formatted in the same way as Figure 3.11.
cases rather than rely on the matrix multiply algorithm.

### 3.6 Acknowledgments

We thank Nils Deppe and Mark Scheel for helpful discussions. Calculations were performed with the SpEC-code [3]. We gratefully acknowledge support from NSERC of Canada, form the Canada Research Chairs Program, and from the Canadian Institute for Advanced Research. Some calculations were performed at the SciNet HPC Consortium [199]. SciNet is funded by: the Canada Foundation for Innovation (CF) under the auspices of Compute Canada; the Government of Ontario; Ontario Research Fund (ORF) – Research Excellence; and the University of Toronto.
4.1 Introduction

Partial differential equations that involve vectorial or tensorial quantities are very common in science. For example, the vacuum Maxwell’s equations,

\[
\begin{align*}
\partial_t \vec{E} &= \nabla \times \vec{B}, \\
\partial_t \vec{B} &= -\nabla \times \vec{E}, \\
\nabla \cdot \vec{B} &= 0, \\
\nabla \cdot \vec{E} &= 0,
\end{align*}
\]

involve manipulations of the vector fields \(\vec{E}\) and \(\vec{B}\). Numerically, these fields are represented by arrays of three numbers per point on a discretized spatial grid. Manipulations of the fields using a language such as C or FORTRAN will involve loops over each component and over the grid-size.

When solving Einstein’s equations of general relativity [51], the necessary tensorial equations can become quite involved. As a moderate example consider the evolution equation of the spatial metric in certain formulations of Einstein’s equations,

\[
\partial_t g_{ij} = -2\alpha K_{ij} + \nabla_i \beta_j + \nabla_j \beta_i, \quad i, j = 1, 2, 3.
\]

Here, \(g_{ij}\) and \(K_{ij}\) are the spatial metric and the extrinsic curvature; both these are represented by spatially varying, symmetric 3x3 matrices. The scalar quantity \(\alpha\) denotes the lapse-function and \(\beta_i\) the shift-vector, both of which are also spatially varying. And finally, \(\nabla_i\) denotes the covariant derivative operator compatible with \(g_{ij}\). Equations (4.1)–(4.5) depend on one or two indices, respectively. Intermediate expressions in general
relativity can easily depend on more indices, for instance the Christoffel-symbols are defined as
\[
\Gamma^i_{jk} = \frac{1}{2} \sum_{l=1,2,3} g^{il} (\partial_k g_{jl} + \partial_j g_{lk} - \partial_l g_{jk}), \quad i, j, k = 1, 2, 3. \tag{4.6}
\]
where \(\partial_i\) denotes the partial derivative, and the 3x3 symmetric matrix \(g^{ij}\) is the inverse of the matrix \(g_{ij}\), both of which are spatially varying. Because of the symmetry in the index-pair \(jk\), Eq. (4.6) represents 18 independent equations, each one with nine terms on the right-hand side.

Henceforth, we adopt the Einstein sum-convention that repeating indices are being summed over (i.e. we will no longer write \(\sum_i\) in equations like Eq. (4.6)). Furthermore, Latin lower-case letters from the middle of the alphabet \((i, j, k, \ldots)\) will range over the three spatial dimensions.

Upon spatial discretization, each spatially dependent tensor is represented on a spatial grid or, for multi-domain methods, on multiple spatial grids. On each such grid, assumed to have \(N\) points, Eq. (4.5) would then, schematically, be represented by code such as that in Listing 4.1.

Listing 4.1: Schematic implementation of Eq. (4.5)

```
Tensor<DataMesh> dtg, K, db;
DataMesh alpha;
// initialize dtg, K, db, alpha
for (int i=0; i<3; ++i) {
   for (int j=0; j<i; ++j) {
      for (int a=0; a<N; ++a) {
         dtg(i, j)[a] = -2*alpha[a]*K(i, j)[a] + db(i, j)[a] + db(j, i)[a];
      }
   }
}
```

The schematic Listing 4.1 indexes tensorial objects with parentheses for the tensor-indices; the grid-points of the underlying grid are indexed with square brackets. We furthermore assume in Listing 4.1 that the covariant derivative of \(\beta_i\) was already precomputed\(^1\) into the variable \(db\).

\(^1\)The covariant derivative is given by
\[\nabla_j \beta_i = \partial_j \beta_i - \Gamma^k_{ij} \beta_k,\]
where the last term in this expression uses the sum-convention.
Our focus in this chapter is the numerical relativity code SpEC, a mature code in active use for the computation of gravitational waveforms for ground-based detectors. Expressions such as that of Listing 4.1 are ubiquitous in SpEC and present a major challenge to development, adaptation, and maintenance. The library presented in this chapter, TLoops, removes from SpEC the need for explicit source-code loops over tensor-indices. Equation (4.5) can then be written as a single line, as illustrated in Listing 4.2.

Listing 4.2: Implementation of Eq. (4.5) in SpEC with the implicit tensor-loop functionality presented in this chapter.

\[
\begin{align*}
\text{Tensor<DataMesh> } & \ dtg, \ K, \ db; \\
\text{DataMesh } & \ alpha; \\
// \ \text{initialize } dtg, \ K, \ db, \ alpha \\
& \ dtg(Sym<0,1>(), i, j) = -2*alpha*K(i, j)+db(i, j)+db(j, i);
\end{align*}
\]

The variables \(i, j, \) etc, are pre-defined by TLoops. Overloaded indexing-operators and assignment-operators are defined such that the single line in Listing 4.2 expands to all relevant loops, both over tensor-indices and over grid-points. TLoops also handles sums. For instance Eq. (4.6) can be coded as the single expression in listing 4.3.

Listing 4.3: Implementation of Eq. (4.6) in SpEC with the implicit tensor-loop functionality presented in this paper.

\[
\begin{align*}
\text{Tensor<DataMesh> } & \ Gamma, \ Invg; \\
\text{Tensor<Tensor<DataMesh>> } & \ dg; \ // \ \text{partial k g}_{ij} = dg(i,j)(k) \\
// \ \text{initialize Gamma Invg, dg} \\
& \ Gamma(Sym<1,2>(), i, j, k) = \\
& \ 0.5*\text{Sum}(l, \ Invg(i, l)*(
\begin{align*}
\text{dg}(j, l)(k)+\text{dg}(l, k)(j) & -\text{dg}(j, k)(l))
\end{align*}
\));
\end{align*}
\]

There already exist several packages implementing similar functionality [1, 38, 185, 253, 306, 314]. Consistent with our observations, benchmarks of them show impaired performance relative to explicitly coded loops [2], presumably due to compiler optimizations being oriented towards the latter.

The true (and to our knowledge unique) advantage of our package is its ability to automatically generate equivalent source code to templated expressions. When a certain compiler flag is defined, TLoops stores each unique tensor expression it encounters within the linker code of each compiled library. A packaged executable, CodeWriter, thus has access to the full list of possible tensor expressions, from which it generates legal non-templated code performing equivalent operations. We present here two examples, a (loop based) C-implementation, and a GPU (CUDA) implementation. In either case,
the original C++ template-code does not need any source-code modifications – the new C- or CUDA-code is incorporated at link-time.

Because of the latter functionality, TLoops can be used to immediately port large numbers of tensor operations to the GPU without the need to explicitly write kernels. These tensor operations are normally substantially faster than CPU code, and allow data to be kept on the GPU between calls to other GPU kernels, allowing segments of code to be hand-ported without extraneous CPU-GPU synchronizations.

The remainder of this chapter is divided into three parts: First, we introduce SpEC and outline the C++ template techniques that enable the compact code in listings 4.2 and 4.3. Second, we present our techniques to allow replacement of the template-generated code with automatically generated non-templated code. Finally, we show detailed benchmarks of the new results.

4.2 Direct evaluation of tensor loops using expression templates

4.2.1 Spectral Einstein Code

The code presented in this paper is based on the Spectral Einstein Code (SpEC) [3] written in C++. In SpEC, arrays over grid-points are represented by the class DataMesh, and tensorial objects are represented by template<class T> class Tensor. SpEC’s class DataMesh already contains expression-templates that handle loops over grid-points. Therefore, in SpEC, Listing 4.1 is coded as

Listing 4.4: Implementation of Eq. (4.5) in SpEC.

| Tensor<DataMesh> dtg, K, db; |
| DataMesh alpha; |
// initialize dtg, K, db, alpha |
| for(int i=0; i<3; ++i) { |
| for(int j=0; j<i; ++j) { |
| dtg(i,j) = -2*alpha*K(i,j) + db(i,j) + db(j,i); |
| } |
| } |

Indexing a Tensor<T>, e.g. K(i,j), returns a (const or non-const) reference to T. SpEC’s class Tensor is aware of symmetric indices. For instance, if K is initialized as symmetric, K(i,j) and K(j,i) both return a reference to the same element.
SpEC’s class DataMesh implements automatic resizing when assigned to. Furthermore, when used on the right-hand-side of assignments (as in Listing 4.4), DataMesh checks consistency of the sizes of all DataMesh’es involved. These consistency checks, combined with the absence of the explicit loop over grid-points and indexing of grid-points already significantly reduces the possibility of coding errors. However, two major shortcomings remain:

1. Loops over tensor indices must be coded manually, which is tedious and error-prone. Specifically, the loops over indices must be consistent with the symmetries of the respective Tensor (cf. the loop over j in Listing 4.4).

2. The existing SpEC expression templates operate on class DataMesh’es. For each combination (i,j), the inner loop thus represents an independent expression on DataMesh, triggering a full traversal of all gridpoints. In Listing 4.4 this requires six traversals of the associated memory, whereas Eq. (4.6) would require 18 traversals.

TLoops corrects both these shortcomings by removing the need to write explicit loops entirely. This is done using a hierarchy of expression templates to represent tensorial manipulations. In the immediately following sections we detail the specifics of those templates.

### 4.2.2 Tensors and SpEC’s Tensor class

For our purposes, tensors are objects with \( R \) indices, each taking \( D \) distinct values. The integer \( R \geq 0 \) is called the rank of the tensor, and \( D \) its dimension. For instance, \( g_{ij}(i, j = 0, 1, 2) \) indicates a rank \( R = 2 \) tensor of dimension \( D = 3 \). If a tensor is symmetric on a pair of indices, then the ordering of the two indices in the pair is irrelevant. For instance, \( \Gamma^i_{jk} \) defined in Eq. (4.6) is symmetric on its two lower indices, i.e. \( \Gamma^i_{jk} = \Gamma^i_{kj} \) for any values of \( i, j, k = 0, \ldots, D - 1 \). The significance of the index-placement (up/down) is irrelevant for the purposes of this paper. Many tensors in general relativity are symmetric on some or all indices, including \( g_{ij} \) and \( K_{ij} \) in Eq. (4.5). In differential geometry tensors must satisfy additional conditions related to coordinate transformations, which are not satisfied by Christoffel symbols \( \Gamma^i_{jk} \). While \( \Gamma^i_{jk} \) are not tensors in the mathematical sense, they are nevertheless represented in SpEC with class Tensor.

In general relativity one commonly encounters both space-time and spatial tensors. Indices of space-time tensors range over the three spatial dimensions and time. An
example is the space-time metric,

$$\psi_{ab}, \quad a, b = 0, 1, 2, 3,$$  \hspace{1cm} (4.7)

where we use letters from the start of the alphabet \((a, b, c, \ldots)\) to indicate space-time indices. The zero-th index-value (e.g. \(a = 0\)) indicates the time-dimension, while \(a = 1, 2, 3\) indicate the space dimensions. The spatial metric \(g_{ij}\) is a subset of the space-time metric,

$$g_{ij} = \psi(i+1)(j+1), \quad i, j = 0, 1, 2.$$  \hspace{1cm} (4.8)

Because SpEC always indexes starting with 0, Eq. (4.8) must add 1 to the spatial indices to obtain the relevant components of \(\psi_{ab}\).

SpEC’s \texttt{Tensor} class represents multi-index objects whose indices each take on the values \(0, 1, \ldots D - 1\). The represented objects may be symmetric on some of their indices, like \(g_{ij}\) or \(\Gamma^i_{jk}\). Internally, a \texttt{Tensor\langle X\rangle} holds an array of elements, each an object \(X\), of appropriate size given the symmetries (e.g. the symmetric \(D = 3\) tensor \(g_{ij}\) has six elements). A \texttt{Tensor\langle X\rangle} furthermore holds a look-up table to translate indices \((i,j)\) into the actual storage location inside the array. Symmetries are implemented by the lookup table for \((i,j)\) and \((j,i)\) pointing to the same element. \texttt{Tensor} is indexed with parentheses, i.e. \texttt{Gamma(0,1,2)} represents \(\Gamma^0_{12}\). Listing 4.5 demonstrates some indexing-operations performed on \texttt{Tensor}, while Listing 4.4 already demonstrated actual computations performed with \texttt{Tensor}.

Listing 4.5: Some typical \texttt{Tensor}-operations in SpEC.

```c
Tensor\langle DataMesh\rangle g, psi, beta;
// initialize g and beta with D=3, and psi with D=4.
// Rank and symmetries as in main text
const int D=g.Dim();
for (int i=0; i<3; ++i) {
    for (int j=0; j<=i; ++j) {
        g(i,j)=psi(i+1, j+1); //(*)
    }
}
for (int i=0; i<D; ++i) {
    beta(i) = psi(i+1, 0);
}
```

The Listings 4.4 and 4.5 use \texttt{class DataMesh}, another SpEC-specific class. DataMesh
represents a multi-dimensional rectangular array, holding one double per grid-point, with dimension \( D \geq 1 \), extents \((N_0, N_1, \ldots, N_{D-1})\) and size \( N = N_0 \cdot N_1 \cdots N_{D-1} \). \textsc{SpEC} implements expression templates: arithmetic operators between \texttt{DataMesh}-objects and/or \texttt{double}'s are overloaded to return recursively defined types encoding the operation and the data type of the operands (\texttt{DataMesh} or \texttt{double}). The instantiations of the expression templates furthermore collect references to the memory locations of all involved data. The assignment operator then recurses through the template to evaluate the expression.

Certain design choices of \textsc{SpEC} present challenges for the development of TLoops. Because \textsc{SpEC} is a well-established and intensely used code, these choices cannot be changed and we have to work within them:

- Dimension, rank and symmetry of a \texttt{Tensor<X>} are assigned dynamically at runtime, and not statically through template-arguments at compile-time. This gives flexibility when using instances of \texttt{Tensor}, because dimension/rank/symmetry can be changed as needed. Unfortunately, this also implies that dimension/rank/symmetry are not available to C++’s type-system at compile-time. Part of this paper therefore deals with injecting compile-time information into the tensor-expressions (e.g. Listings 4.2–4.5) so that the information needed to construct loops over tensor-indices can be deduced at compile-time.

- \texttt{Tensor<T>} is a template class which stores an array of \( T \)'s. Because each \texttt{DataMesh} allocates its own storage independently, this implies that \texttt{Tensor<DataMesh>} has independent \texttt{double*} arrays of size \( N \) for each tensor-component, rather than one contiguous array of size \( N_{\text{components}} \times N \). While \textsc{SpEC}'s design-choice makes it convenient to use Tensors, it is not necessarily computationally optimal. Specifically, for GPU-implementations of tensor-loops, the increased number of memory locations degrades performance, cf. Section 4.5.

- Finally, because of \textsc{SpEC}'s age, and the need to run on various supercomputers with varying degree of up-to-date compilers, \textsc{SpEC} restricts itself to C++03 with only a small set of newer C++11 features, and no C++17-specific features.

### 4.2.3 Capturing a TLoops-expression as a type

#### Classes for indexing a Tensor

All expression templates begin with capturing the structure of the expression as a \texttt{type}. Types are available to the compiler and thus enable meta-programming at compile-time. In this section we detail the classes we have developed to accomplish this.
Two classes represent indexing with tensor-indices, i.e. the variables \(i, j, \ldots\) appearing in tensorial equations like Eqs. (4.5) and (4.8): class \(\text{TIndex}\) represents an index (e.g. \(i\)) across the entire expression, whereas class \(\text{TIndexSlot}\) is specific to each occurrence of \(i\).

Listing 4.6: Classes \(\text{TIndex}\) and \(\text{TIndexSlot}\).

```cpp
template<int dim, int label>
class TIndex {

public:
    static int Value() const { return mValue; }
    static bool Done() const { return mValue >= dim; }
    static bool Increment() const { ++mValue; }
    static bool Reset(const int ctr=0) { mValue=ctr; }

private:
    static int mValue;
};

template<int dim, int label, int offset>
class TIndexSlot: public TIndex<dim, label> {

public:
    static int Value() { return TIndex<dim,label>::Value()+offset; }
    static const int Offset=offset;
};

// define variables for tensor-loop expressions
// 3-dimensional indices (incomplete list)
extern TIndexSlot<3,0,0> i_;
extern TIndexSlot<3,1,0> j_;

// 4-dimensional indices (incomplete list)
extern TIndexSlot<4,0,0> a_;
extern TIndexSlot<4,1,0> b_;```

Listing 4.6 defines two classes and a set of variables. class \(\text{TIndex<dim, label}>\) serves as a marker to enable type-capture. As such, \(\text{TIndex}\) is templated on the dimension of the index. The additional \(\text{label}\)-argument distinguishes indices of the same dimension.
Chapter 4. Tensor Loops

(e.g. \(i, j, \ldots\).) \texttt{TIndex} also contains a counter “mValue” and functionality to iterate this counter over the allowed index-values of the given \texttt{TIndex}. This functionality will become relevant when we discuss evaluation of a tensor-expression in Section 4.2.4.

\texttt{class TIndexSlot<\texttt{dim, label, offset}>} tags each individual occurrence of an index in an expression. This is required because different occurrences can have different offsets (i.e. “\(i\)” and “\(i+1\)”), which therefore require different encodings. \texttt{TIndexSlot} inherits from \texttt{TIndex}, in order to allow easy down-casting, which is convenient for compile-time consistency checks of the index structure.

Finally, Listing 4.6 defines variables \(i_-, j_-, \ldots\), that map to specific \texttt{TIndexSlots}. These enable the user to inject type-information about indexing into source code, conveniently resembling mathematical expressions in tensor calculus.

**Types representing an indexed Tensor**

The classes and variables defined in the previous subsection (\texttt{TIndex}, \texttt{TIndexSlot}, \(i_-, \ldots\)) are used to index a tensor, e.g. \(g(i_-, 1)\). This is accomplished with a suitable \texttt{Tensor::operator()}, which will return a type that carries all information about the indexing. This return-type is built from helper classes introduced schematically in Listing 4.7. The first two helper classes (\texttt{TInequality} and \texttt{TSymmetry}) handle symmetries of tensors.

The marker-class \texttt{TInequality<\texttt{pos1,pos2}>} indicates a symmetry between \textit{one} pair of indices in a tensor, namely the indices at \texttt{pos1}'th and \texttt{pos2}'th position (where the counting starts from zero). For example, consider the metric \(g_{ij}\) and its time-derivative \(\partial_t g_{ij}\) both of which are symmetric in their only two indices. When iterating over all components of these tensors, this implies a condition \(j \leq i\), cf. Listing 4.4. \texttt{TInequality<0,1>} precisely represents this inequality, in that it indicates that in loops, the value of the 0-th index should be equal or larger than the value of the 1-th index.

A generic tensor with any structure of symmetric indices\(^3\) can then be represented by a set of \texttt{TInequality}'s. For a tensor without symmetries, this set is empty. For a tensor with one symmetric pair of indices, the set contains one \texttt{TInequality}, like for \(g_{ij}\) or for \(\Gamma^i_{jk}\) (\texttt{TInequality<1,2>}). More generic symmetries are represented by multiple \texttt{TInequality}'s. For instance, a rank-four tensor \(A_{ijkl}\) separately symmetric on the first two and last two indices is represented by \texttt{TSymmetry<TInequality<0,1>, TInequality<2,3>>}, and a completely symmetric rank-three tensor \(B_{ijk}\) by \texttt{TSymmetry<TInequality<0,1>, TInequality<1,2>>>}.

\(^3\)Recall that we do not consider tensors with anti-symmetric pairs of indices, or with cyclic symmetries like those of the Riemann tensor.
Such sets of inequalities are represented by class `TSymmetry<...>`, which is only defined in specializations for `TInequality<.,.>`, and which utilizes compile-time asserts to enforce a monotonically increasing ordering of the `TInequality`'s.

Creation of the `TSymmetry<...>` marker-classes is handled via free template functions `Sym<i,j>()`, `Sym<i,j,k>()`, `Sym<i,j,k,l>()` that return the `TSymmetry` class representing full symmetrization on the indicated slots. For the case of several distinct symmetries, e.g. $A_{ijkl}$, operator `&` is suitably overloaded to allow `Sym<0,1>() & Sym<2,3>()`.

Knowledge of the symmetries of a tensor is only important on the *left-hand side* of an assignment, as only the indices on the left-hand side are looped over. Therefore, it is optional to specify symmetries for tensors on the right-hand side, as in Listing 4.2. Doing so does not throw an error, but also has no effect.

Listing 4.7: Classes representing an indexed Tensor, i.e. its symmetries and how it was indexed.

```cpp
template<int pos1, int pos2>
struct TInequality {
    static_assert(pos1<pos2,
        "TInequality must satisfy pos1<pos2" );
};

template<class ...TInequalities> struct TSymmetry;

template<class TSymmetry_t, class ...indices>
struct TIndexStructure;
```

With symmetries of a tensor handled by `TSymmetry`, we now turn to the indexing of a tensor. This is handled by class `TIndexStructure<TSymmetry<...symm>, ...indices>`, where the template-pack `indices` has a length equal to the number of indices. Each type in this template-pack is either a `TIndexSlot` or an `int`. The former indicates indexing with an implicit index (as used in Listing 4.2), whereas the latter case indicates the usual indexing by an integer (as used in Listing 4.4). Indexing with implicit indices and integers can be mixed. Consider, for example, a rank 3 tensor symmetric on the last two indices, which is indexed on the first slow with the integer 1. In mathematical notation, this is represented by $\Gamma_{ij}^1$, which is mapped by `TLoops` to `Gamma(1,i-, j-)` with indexing structure `TIndexStructure<TSymmetry<TInequality<1,2>>, 1, TIndexSlot<3,0,0>, TIndexSlot<3,1,0>>`. 
The classes `struct TSymmetry<...TIneq>` and `TIndexStructure<TStructure_t>, ...indices>` are recursively defined in the number of `TInequality`'s and tensor-indices. The specializations for the empty case are trivial. One then adds additional inequalities/indices at the front via variadic template arguments. Each specialization defines several member types and member-variables that will be useful subsequently, and which are shown in Listing 4.8.

Listing 4.8: Member types and variables for `TSymmetry` and `TIndexStructure`, which will be used in the automatic generation of the implicit loops.

```cpp
template<class ...TInequalities>
struct TSymmetry {
    // TSymmetry for tensor with first slot removed
    typename Shift_t;
};

template<class TSymmetry_t, class ...indices>
struct TIndexStructure {
    // === ADMINISTRATION ===
    // rank of tensor (# of free indices + # of indexed indices)
    static const int Rank;
    // number of free indices
    static const int NFree;
    // number of distinct free indices
    static const int NUniqueFree;
    // set of all free indices
    typename TindexSet_t;
    // TIndexStructure with 1st slot removed
    typename BASE;
    
    // === ITERATION ===
    // increment to next set of free indices
    void operator++();
    // iteration over free indices complete?
};
```
operator bool() const;
// reset iteration over all free indices
void Reset();

//=== ACCESS ====

// retrieve value of first index
int GetFirstIndex() const;

// retrieve value of N-th index (zero-counted)
template<int N>
int GetNthIndex() const;

// retrieve values of all indices as a size=Rank vector
void GetAllIndices(MyVector<int>& idx) const;

The member types and variables of TSymmetry and TIndexStructure are used in subsequent steps to implement functionality. For instance, when adding two indexed tensors, the suitable operator+ will static assert that both indexed tensors have the same set of free indices. This mirrors mathematical meaning, where \( g_{ij} + \beta_i \beta_j \) is correct, whereas \( g_{ij} + \beta_i \beta_k \) is erroneous.

Expression-tree for implicit tensor loop expressions

TIndexStructure represents the full indexed structure of an indexed tensor, i.e. the information needed to prepare at compile-time (via metaprogramming) the necessary loops. In order to execute the operation, in addition, the memory locations of all Tensor<DataMesh> instances are needed. The memory locations will be stored in a recursive expression-tree assembled with a template class iBinaryOp<L, Op, R> taking three template-parameters: The first template-parameters \( L \) and \( R \) represent the left and right operands. These operands could be of type iBinaryOp themselves, thus enabling the recursion to represent nested expressions. The middle template-parameter \( Op \) represents the mathematical operation to be performed.

The 'i' in iBinaryOp indicates that the relevant expression is indexed by at least one symbolic tensor-index (\( i, j, ... \)). This is important, because for tensorial expressions, only a small subset of mathematical operations are permissible: (i) Addition and subtraction, for which the tensor-indices in both operands must agree; (ii) Multiplication; and (iii) negation. In contrast, SpEC expression-templates operating on DataMesh utilize...
a much larger set of mathematical operators, including division, and a greatly enhanced set of unary operators (like square-root and trigonometric functions).

There are several groups of partial specializations of \texttt{iBinaryOp}, to enable its full functionality. These specializations are schematically indicated in Listing 4.9.

Listing 4.9: Schematic specializations of \texttt{iBinaryOp}.

// Set (1): Two indexed expressions
\texttt{iBinaryOp<iBinaryOp,MultOp,iBinaryOp>}
\texttt{iBinaryOp<iBinaryOp,AddOp,iBinaryOp>}
\texttt{iBinaryOp<iBinaryOp,SubOp,iBinaryOp>}
\texttt{iBinaryOp<EmptyType,negateOp,iBinaryOp>}

// (2a) One indexed expression and one double
// \( d\times\texttt{iOp}, \texttt{iOp}\div d \)
\texttt{iBinaryOp<\texttt{double},MultOp,iBinaryOp>}
\texttt{iBinaryOp<iBinaryOp,MultOp,\texttt{double}>}
\texttt{iBinaryOp<iBinaryOp,DivOp,\texttt{double}>}

// (2b) One indexed expression and one DataMesh
// \( \text{DM}\times\texttt{iOp}, \texttt{iOp}\div \text{DM} \)
\texttt{iBinaryOp<DataMesh,MultOp,iBinaryOp>}
\texttt{iBinaryOp<iBinaryOp,MultOp,DataMesh>}
\texttt{iBinaryOp<iBinaryOp,DivOp,DataMesh>}

// (2c) One indexed expression and one scalar-valued
// DataMesh expression
// \( \text{BOp}\times\texttt{iOp}, \texttt{iOp}\div \text{BOp} \)
\texttt{iBinaryOp<BinaryOp,MultOp,iBinaryOp>}
\texttt{iBinaryOp<iBinaryOp,MultOp,BinaryOp>}
\texttt{iBinaryOp<iBinaryOp,DivOp,BinaryOp>}

// (3) leaf-node: one indexed Tensor<DataMesh>
\texttt{iBinaryOp<TIndexStructure<TSymmetry<Sym...>, Indices...>, EmptyType,DataMesh>
Set (1) contains the recursive operators that combine two indexed expressions together. As explained above, mathematically there are only four allowed operators which are represented by the marker-classes \texttt{AddOp}, \texttt{SubOp}, \texttt{MultOp} and \texttt{negateOp}. For instance the '+' operators in Listing 4.2 are represented by \texttt{iBinaryOp}'s of set (1). \texttt{iBinaryOp<EmptyType,negateOp,iBinaryOp> } illustrates our convention to indicate unary operators with the type \texttt{class EmptyType }\{ \} in lieu of the first template argument \texttt{L} to \texttt{iBinaryOp}.

Sets (2) recursively combine an indexed expression with a scalar expression (either a 	exttt{double}, or a \texttt{DataMesh}, or a (scalar-valued) expression template of \texttt{DataMesh}, represented by \texttt{class BinaryOp}. Only multiplication and division is mathematically permissible, leaving only three cases each. The multiplication \texttt{0.5*... } in Listing 4.3 is represented by a specialization of set (2a), and the term \texttt{2*N*K(i\_0, j\_0) } is represented by a specialization of set (2c), combining the \texttt{DataMesh}-expression \texttt{2*N} with the indexed expression \texttt{K(i\_0, j\_0)}.

Set (3) is the entry point into the recursive \texttt{iBinaryOp}-representations; it represents one indexed \texttt{Tensor<DataMesh}. Examples of this type include \texttt{K(i\_0, j\_0) } in Listing 4.2 as well as \texttt{psi(i\_0+1, 0) } which arises when rewriting the last loop of Listing 4.5 in implicit tensor notation.

The construction of recursive \texttt{iBinaryOp}s is handled by the relevant overloaded operators. The recursive types of set (1) and (2) are returned by suitably defined \texttt{operator+}, \texttt{operator-}, and \texttt{operator*}. Leaf-nodes of set (3) are returned by suitably templated \texttt{Tensor<DataMesh>::operator()}, which are provided in two versions: with and without a first argument of type \texttt{TSymmetry}. Such a \texttt{TSymmetry} argument must be provided on the \texttt{left-hand-side} of assignments, like \texttt{dg(Sym<0,1>(), i\_, j\_)} in Listing 4.2, because the symmetry is required to determine the loop-bounds. The instance of \texttt{TSymmetry} passed into \texttt{Tensor<DataMesh>::operator()} is encoded in the templated return-type of this operator within the \texttt{TIndexStructure}-parameter inside the set (3) type in Listing 4.9. On the \texttt{right-hand-side}, compile-time information about the symmetry of the tensors is not needed and therefore, presently, it is optional to specify the symmetry through an extra first argument to \texttt{Tensor<DataMesh>::operator()}\textsuperscript{4}.

All \texttt{iBinaryOp} specializations have certain member-types and member-variables which are useful when assembling the types recursively, and when evaluating the implicit tensor-loop expression. These members are indicated in Listing 4.10.

\textsuperscript{4}All examples in this paper omit the symmetry specifiers on the right-hand-side.
Listing 4.10: Member types and variables of iBinaryOp.

```cpp
class iBinaryOp< L, Op, R> {

    // member types
    using TIndexSet_t = ...;
    using ExpandIndices_t = ...;

    // member variables: references to sub-expressions
    const L& lhs;  // (absent for unary operators)
    const R& rhs;
}
```

`TIndexSet_t` is a template-type that represents the set of all free indices; this set is used in `operator+` and `operator-` to verify that both operands have the same free indices. `ExpandIndices_t` is the `DataMesh`-expression type that results when all implicit tensor-indices are replaced by concrete values, i.e. when each `Tensor<DataMesh>` is replaced by the `DataMesh` of one of its components. This type will be used when evaluating the tensor-loop expression, cf. Sec. 4.2.4. Finally, the references `lhs` and `rhs` are also needed during evaluation of the tensor-loop expression, as they contain the concrete memory locations of all relevant data.

### 4.2.4 Evaluation of TLoops-template

The preceding sections describe the individual elements that make up an implicit-tensor loop assignment, as in Listing 4.2: The left-hand-side of this expression expands to a `iBinaryOp` of `Set(3)`, whereas the right-hand-side expands to a `iBinaryOp` of arbitrary complexity. These two elements are combined via the member-assignment operator `operator=( )` of the left-hand-side’s type.

Listing 4.11: Assignment operators for implicit tensor loops.

```cpp
template<class ...Symm, class ...Indices>

class iBinaryOp<TSymmetry<Symm...>, Indices...>,
    EmptyType, DataMesh> {

    // iBinaryOp on right-hand-side
    template<class L, class O, class R>
    void operator=(const iBinaryOp< L, O, R>& op) {
```
CheckIndexEquality(op);
CheckExtentsAndResizeToMatch(op);
TLoopApply(*this, TSetEqualOp(), op);
}

// just a double on right-hand-side (e.g. to set to zero)
void operator=(const double d) {
    TLoopApply(*this, TSetEqualOp(), d);
}

// DataMesh on right-hand-side
void operator=(const DataMesh& dm) {
    CheckExtentsAndResizeToMatch(dm);
    TLoopApply(*this, TSetEqualOp(), dm);
}

// repeat for +=, -=
template<class L, class O, class R>
void operator+=(const iBinaryOp<L,O,R>& op) {
    CheckIndexEquality(op);
    CheckExtentsAndResizeToMatch(op);
    TLoopApply(*this, TAddEqualOp(), op);
}

Listing 4.11 indicates the structure of these assignment operators. There are several such assignment operators depending on the type of operation (=, +=, -=, *=, /=) and depending on the right-hand-side type (iBinaryOp, DataMesh, double). Only some combination of these are mathematically permissible, and only those are defined. As appropriate, these operators check that free tensor-indices on the left-hand-side and the right-hand-side match, and they resize the data on the left-hand-side. Then all these operators call a templated free function TLoopApply for the actual computations. This allows us to handle the different types of assignment (=, +=, -=, ...) without code-duplication.
Listing 4.12: Assignment of implicit tensor loops.

```cpp
template<class L, class O, class ApplyOp, class RHS>
void TLoopApply(iBinaryOp<iBinaryOp<L, O, DataMesh>>& lhs,
                 const ApplyOp&,
                 const RHS& rhs) {
    lhs.CheckUniqueIndices();
    lhs.CheckSymmetries();
    for (lhs.Reset(); lhs; ++lhs) {
        BinaryOpHolder<RHS> holder(rhs);
        ApplyOp::modify(lhs.ExpandIndices(), holder.op);
    }
}
```

Listing 4.12 executes the actual calculations, and as such requires detailed explanations:

1. TLoopApply() starts with safety checks: CheckUniqueIndices is a compile-time check that there are no repeated tensor-indices on the left-hand-side. This test catches, for instance, the typo “dg(Sym<0,1>(), i, i)” in the left-hand-side of Listing 4.2, which is mathematically forbidden. CheckSymmetries() verifies that the stated symmetries in the assignment —e.g. Sym<0,1>() in Listing 4.2— agree with the run-time symmetry-state of the respective tensor. Because of SpeC’s design decision that symmetries of Tensor<X> are set at run-time, this test necessarily can only trigger run-time errors.

2. The loop for(lhs.Reset(); lhs; ++lhs) forwards directly to the corresponding member-functions of TIndexStructure shown in Listing 4.8. TIndexStructure uses recursive template-pack expansion to recurse through all tensor-indices. The loop will modify the static member-variables int TIndex<dim, label>::mValue of the TIndex-types occurring on the left-hand-side, cf. Listing 4.6. During the ++lhs increment, these variables will be reset whenever they reach this upper bound. In this case, the TInequality parameters indicate the position of a potential other index with which an inequality (arising from a tensorial symmetry) must be satisfied. If so, int TIndexStructure::GetNthIndex<int>() retrieves the current value of this other index, which is used in re-setting the index under consideration. Overall, the assignment dg(Sym<0,1>(), i, j)=... in Listing 4.2, results in loops
for(j=0; j<3; ++j) { for(i=j; i<3; ++i) { ... } } 
where ‘i’ represents TIndex<3,0>::mValue and ‘j’ represents TIndex<3,1>::mValue.

3. Inside the loop in Listing 4.12, we must now index each Tensor<DataMesh> on the right-hand-side ‘rhs’ with the current set of index-values as stored inside the respective TIndex<dim,label>::mValue. Upon such indexing, each Tensor<DataMesh> in the right-hand-side expression becomes a standard SpEC_DataMesh, and the expression-tree becomes a regular DataMesh-expression template tree of type RHS::ExpandIndices_t (cf. Listing 4.10). The helper-class BinaryOpHolder recursively descends through ‘rhs’’s structure, and builds an instance of the DataMesh-expression with all data-references pointing to the appropriate elements of each Tensor<DataMesh>.

4. Finally, the actual assignment happens in ApplyOp::modify(). This member-function of the marker-classes SetEqualOp, PlusEqualOp, MinusEqualOp takes its second argument (i.e. the DataMesh expression template representation), and assigns/adds/subtracts it from its first argument (the DataMesh returned by indexing the left-hand-side with the current set of tensor-indices), thus triggering execution of SpEC_DataMesh expression template code.

4.2.5 Sum-operations

Let us now turn to an exposition of contractions as in Listing 4.3.

The goal is to transform the expression $\text{Sum}(k_, op[k_])$ (schematically) into the expression $\text{op}[0]+\text{op}[1]+\text{op}[2]$. Here ‘op’ indicates a tensor-loops expression which may have an arbitrary number of free indices. These should remain intact in the output expression. In our code, this is implemented with a template-class PartialSum<curr_dim, TIndex_t, iBinaryOp_t>. An instance of this class is responsible for handling the index-value curr_dim of the tensor-index TIndex_t. This class recursively decrements curr_dim via inheritance of PartialSum<curr_dim-1, TIndex_t, iBinaryOp_t>, and during recursion assembles the full sum. The corresponding code is shown schematically in Listing 4.13.

Listing 4.13: class PartialSum which forms the core of the implementation of Sum.

```cpp
// Recursion:
template<int curr_dim, class TIndex_t, class iBinaryOp_t>
struct PartialSum:
```
public PartialSum<curr_dim−1, TIndex_t, iBinaryOp_t> {

using BASE=PartialSum<curr_dim−1, TIndex_t,iBinaryOp_t>;

// (a) type of unrolled sum
using ExpandIndices_t = BinaryOp<
typename BASE::ExpandIndices_t,
  AddOp,
  typename iBinaryOp_t::ExpandIndices_t>;

// (b) constructor creating the sum-expanded BinaryOpHolder
PartialSum(const iBinaryOp_t& summand):
  BASE(summand),
  this_term((TIndex_t::Reset(curr_dim), summand)),
  partial_sum(BASE::partial_sum, this_term.op)
{ };

const BinaryOpHolder<iBinaryOp_t> this_term;
const ExpandIndices_t partial_sum;
}

// Break recursion (curr_dim=0)
template<class TIndex_t, class iBinaryOp_t>
struct PartialSum<0, TIndex_t, iBinaryOp_t> {
  using ExpandIndices_t=typename iBinaryOp_t::ExpandIndices_t;

  PartialSum(const iBinaryOp_t& summand):
    this_term((TIndex_t::Reset(0), summand)),
    partial_sum(this_term.op)
  { };

  const BinaryOpHolder<iBinaryOp_t> this_term;
  const ExpandIndices_t& partial_sum;
};

In principle, the framework presented here could detect implicit sums even without
the explicit Sum(...), by watching via meta-programming for duplicate TIndex<.,.> in 
operator*. Walter Landry’s FTensor behaves in this way and presents the choice as a 
design feature [185]. We choose not to implement such implicit loop functionality for two 
reasons: First it would leave evaluation order according to C++ operator precedence, 
and it is not guaranteed that C++ precedence rules will result in optimal evaluation. 
The requirement to explicitly place Sum(...) in the code will force the user to make 
an explicit choice of how terms will be grouped, thus exhibiting the FLOPS implications 
more clearly. Second, sums exponentially increase the amount of FLOPs in an expression. 
The explicit occurrence of Sum, especially when repeated multiple times in the same 
expression, acts as signal for potentially very expensive operations.

4.3 Automatic code generation

In this section we describe TLoops’ automatic code generation functionality, which represents 
TLoops expressions with equivalent C or CUDA code. First, SpEC is compiled with 
certain options what encode each TLoops expression it uses. Next, an executable called 
CodeWriter iterates through the encoded expressions and outputs new code for each. 
SpEC is then recompiled with this new code, which replaces the expression-templates 
described in Section 4.2 at link-time. This gives in total four different SpEC compilation 
variants: NonAccel, as always; CodeWriter, to output the new equivalent code; 
AccelCPU, to link in the automatically-generated C code; and AccelCUDA, to link in the 
automatically-generated CUDA code.

Let us first give a high-level overview of the tools which generate this code. As 
described in Section 4.2, TLoops expressions are represented as trees. Each node in the 
tree represents either an operator, in which case it has either one or two subnodes, or 
actual data (of type double, DataMesh, or indexed Tensor<DataMesh>), in which case it 
has no subnodes and we call it a “leaf”. The root node represents the type of assignment 
(=, + =, − =, or ∗ =). In Section 4.2 this expression tree is built at compile-time with 
recursive templates, with the one goal of executing the encoded calculation.

In Figure 4.1 we illustrate the tree structure appropriate to the operation \( \partial_i g_{ij} = -2\alpha K_{ij} + \nabla_i \beta_j + \nabla_j \beta_i \). The top panel shows the TLoops source-code expression. The 
second and third illustrate the expression tree. In the second panel that tree is illustrated 
by a shorthand representation of the expression template. BOp, in particular, stands in 
for the iBinaryOp introduced in Listing 4.9 and the surrounding text.

The third panel illustrates the tree recursion performing automatic C code generation. First, we generate an appropriate set of for loops from the index structure of the
LHS tensor. We next iterate through the tree nodes representing operators, outputting
variables that represent concrete data (such as d0 for double) from the child leafs of each
node.

We now turn to equivalent code output in multiple languages (C and CUDA) il-
lustrated in Figure 4.1. Performing this output using compile-time templates proved
cumbersome, since such template-based code is difficult to write and debug, and is too
inflexible for our diverse goals. We therefore have developed a secondary run-time rep-resenta-
tion of the expression tree as concretely-instantiated C++ classes, which works as
follows:

- The abstract class TExpressionLeafBase represents a leaf in the expression tree,
  with concrete derived classes for each type of leaf, such as double, DataMesh, or
  indexed Tensor<DataMesh>.

- The abstract class TExpressionOperatorBase represents operators, with one
  concrete derived class for each (+, −, sqrt, etc).

- class TExpressionNode represents a node in the expression, which may be a leaf
  or an operator. This is the class which forms the actual tree structure, and which
  handles recursion. It holds pointers to any child TExpressionNodes, as well as a
  pointer to the

  TExpressionLeafBase*, if a leaf, or TExpressionOperatorBase*, if an operator.

With the above class-representation of an expression in hand, we are now ready

to output actual code. Conceptually, each class will trigger output of whatever code-
fragment it represents:

- A TExpressionOperatorBase will have member functions to output the string
  representation (‘+', ‘sqrt', etc.). These member functions will place the operants
  in the right places, e.g. on either side of binary operators like ‘+', or within the
  parentheses of unary operators like ‘sqrt()'.

- A TExpressionLeafBase has member functions to output any code fragments di-
  rectly involving the operand. For example, the member function std::string
  VarDeclaration();, which outputs variable declarations for the C-style code, out-
  puts const double d0; in the case of the first double on the right hand side,
  or const double* TDm1[3]; in the case of the second Tensor on the right hand
  side, with a rank of 1 and dimension of 3.
\[ \frac{\partial t}{g_{ij}} = -2\alpha K_{ij} + \nabla_i \beta_j + \nabla_j \beta_i \]

dg(Sym<0,1>(), i, j) = -2.*alpha*K(i,j) + beta(j)(i) + beta(i)(j);

\[
\text{BOp<TIndStr<Sym<0, 1>, Ind<0>, Ind<1>>, EqualsOp}, \\
\text{BOp<double, MultOp,} \\
\text{BOp<DataMesh, MultOp,} \\
\text{BOp<TIndStr<Ind<0>, Ind<1>>, PlusOp,} \\
\text{BOp<std::pair<TIndStr<Ind<1>>, TIndStr<Ind<0>>, PlusOp,} \\
\text{std::pair<TIndStr<Ind<0>>, TIndStr<Ind<1>>>>>;} \\
\]

for(int i=0; i<3; ++i){
    for(int j=i; j<3; ++j){
        for(int x=0; x<GRIDSIZE; ++x){
            TDM0[i][j][x] =
            d0 * 
            D0[x] * 
            TDm1[i][j][x] + 
            TTDm0[j][i][x] + 
            TTDm1[i][j][x];
        }
    }
}

Figure 4.1: Various representations of the \( \partial_t g_{ij} \) operation: in mathematical notation (top), as a TLoops expression (second), as a nested series of TLoops expression templates (third), and as the automatically output code diagrammed with each fragment in the appropriate part of the semantic tree used to represent expressions at runtime (bottom). We use shorthands for the class types in the expression template: BOp for iBinaryOp, TIndSt for TIndexStructure, and Ind for TIndexSlot. Within the tree, DM represents DataMesh (i.e. a single component array), TDm represents Tensor<DataMesh>, and TTDm represents Tensor<Tensor<DataMesh>>.
• A TExpressionNode will have member functions PrintExpression. In the case of a leaf, these call the appropriate code output function from TExpressionLeafBase. In the case of an operator, they call the output functions of the associated TExpressionOperatorBase along with those of the next child TExpressionNode, with parentheses formatted appropriately depending on whether the operator is unary or binary.

So far, we have described the structure and operation of a fully initialized TExpressionTree, and how such a tree yields the desired output code. We now turn to the construction of these TExpressionTrees. First, we must interface the templated representation of an expression (iBinaryOp<L, Op, R>) with this new C++ class-based representation. To do so, we proceed as follows:

• We define a set of C++ functions that are templated on the expression-template representation. These functions call one another recursively, and in this way they recurse through the expression-template representation in the appropriate order, returning at each stage the relevant part of the class-based representation, i.e. the correct TExpressionNodes.

• We further define a templated wrapper class ConcreteTExpressionTreeHouse which constructs the class-representation and which provides convenient functions to interact with it. This class is derived from an abstract base-class TExpressionTreeHouse, which hides the type of the concrete expression-template. Thus, having a pointer to TExpressionTreeHouse, surrounding code can interact with the expression in a type-agnostic way, making it possible to loop over different expressions and output code.

At this stage, we are faced with the task of constructing one ConcreteTExpressionTreeHouse for each distinct expression-type in SpEC, and collecting pointers to the TExpressionTreeHouseBase in one large list, so that we can iterate over the tree. For this, we utilize the existing code in SpEC named Factory, which implements the factory design pattern [127].

Conceptually, for each abstract base-class in SpEC, there is a database called Factory within which each concrete derived class registers itself, providing an ID-string along with a pointer to a function that creates the concrete derived class. After registration, Factory can then be passed ID-strings, causing it to call the relevant create-function and to return a pointer to the newly created instance of the polymorphic class.
We use the SpEC-Factory as follows. When SpEC is compiled with the flag
−DCODE_WRITER, each TLoopApply-template function - the function which triggers eva-
uation of the expression template, and which is thus itself uniquely templated on the
expression - activates extra code that defines a static variable

\texttt{static bool registered\_\_\_\_ = Factory:\:Register(options);} 

In the above, \_\_\_\_ represents a unique string constructed from the complete template
type by recursive calls to a function \texttt{TNameHelper} mapping expression-template types to
string fragments.

During standard execution, these extra variables have no effect. They instead be-
come important when linked into the CodeWriter executable. Upon initialization of the
static variables at the start of CodeWriter execution, each registered variable causes the
relevant call to \texttt{Factory::Register}, thus building a database containing all expression-
templates occurring within the object files. Each entry in this database will now be
represented by a concrete derived class of the CodeWriter base-class, making the list of
expression templates available at runtime to the executable. At that point CodeWriter
iterates through all these concrete derived classes, creates one instance of the expression
tree class-representation from each, and calls the relevant member functions to output C
and CUDA code.


\texttt{void CodeWriter::Write()\{ 
  int fnumber=0; 
  const std::list<std::string> Exps 
  = Factory::RegisteredClassIDs<
      TExpressionTreeHouseBase>(); //*
  for (auto ExpTag:Exps)\{ 
    ++fnumber; 
    TExpressionTreeHouseBase* TreeHouse_ptr = 
    TExpressionTreeHouseBase::
      CreateDerivedClass(ExpTag); //**
    WriteEntry(TreeHouse_ptr, fnumber);
    delete TreeHouse_ptr;
  \}
\}}

Listing 4.14 illustrates CodeWriter’s iteration-through-templates procedure. In that
Listing, the line marked \texttt{/*} retrieves from the Factory a list of all possible ID-strings
which represent derived classes from `TExpressionTreeHouseBase`, and thus which represent expression templates. `CodeWriter` then iterates through that list and, in the line marked `//**`, constructs a concrete instance of `ConcreteTExpressionTreeHouse` for each expression.

`CodeWriter` outputs into three files. The first contains functions whose arguments are templated on the appropriate `TLoops` expression template. These functions route to either the CUDA or the CPU code depending on which of `AccelCPU` or `AccelCUDA` are defined. The second extracts the actual arrays of pointers from the `iBinaryOp`’s passed to the function and passes these on. In the CUDA case this array of pointers must be copied to the GPU via an API call, which can be a significant extra expense. To avoid this we make the copy only once, repeating only if the structure of the `Tensor` changes. The third file contains the actual functions and, in the CUDA case, tuning arguments for the kernels, which are chosen based on the tensor structure of the left-hand side (see Section 4.5 for details).

### 4.4 Design Considerations of Automatically Generated CUDA Code

Section 4.3 detailed the tools we have developed to output automatically generated C and CUDA code to perform `TLoops` operations. In this section we describe the structure of that code with an eye to its performance.

First, let us give some examples of C-style code generation. Consider, for example, the `TLoops` expression in Listing 4.15,

```plaintext
Listing 4.15: TLoops rank 2 contraction.
C(Sym<0,1>() , a_ , b_ ) = Sum( c_ , A(a_ , c_ ) * B( c_ , b_ ) );
```

which symmetrically contracts the rank-2 tensors `A` and `B` over their shared index `c_`. `CodeWriter` generates from Listing 4.15 the C-style code in Listing 4.16.

```plaintext
for (int b=0; b<4; ++j) {
    for (int a=b; a<4; ++a) {
        for (int x=0; x<N; ++x) {
            double sum=0;
            for (int c=0; c<4; ++c) {
                sum+=A[a][c][x]*B[c][b][x];
            }
        }
    }
}
```

Listing 4.16: C-style code corresponding to Listing 4.15.
Chapter 4. Tensor Loops

Recall \( N \) is the spatial gridsize. In Listing 4.16, \( a \) in the \texttt{for} loop marked is initialized to \( b \) rather than to 0, due to the \texttt{Sym<0,1>()} flag in Listing 4.15. The \texttt{for} loops run up to 4 due to the use of \( a_-, b_- \) rather than \( i_-, j_- \) indices, which would generate loops running to 3. Indexing may be further controlled by ‘fixing’ indices (e.g. by specifying an integer value, such as 1, instead of an index such as \( i_- \)), or by specifying index “offsets” such as \( i_-+1 \). Thus, the expression in Listing 4.17 generates the C-style code in Listing 4.18.

Listing 4.17: TLoops rank 2 contraction demonstrating fixed and offset indices.

\[
D(\text{Sym<0,1>()}, i_-, 0) = \text{Sum}(c_-, E(i_-+1, c_-) \times F(c_, 0));
\]

Listing 4.18: C-style code corresponding to Listing 4.17.

```c
for (int i = 0; i < 3; ++i){
  for (int x = 0; x < N; ++x){
    double sum = 0;
    for (int c = 0; c < 4; ++c){
      sum += E[i+1][c][x]*F[c][0][x];
    }
    D[i][0][x] = sum;
  }
}
```

note that in this case the \texttt{Sym<0,1>()} flag has no effect.

Let us now demonstrate our automated CUDA code, starting with a simplified example generated from the expression in Listing 4.19.

Listing 4.19: TLoops rank 2 contraction.

\[
C(a_-, b_-) = \text{Sum}(c_-, A(a_-, c_-) \times B(c_, b_-));
\]

This differs from Listing 4.15 only in that the symmetry flag has been removed. In CUDA, instructions to the GPU are collected into function-like entities called \texttt{kernels}. The kernel generated from Listing 4.19 closely resembles that presented in Listing 4.20.

Listing 4.20: CUDA kernel corresponding to Listing 4.19.
Chapter 4. Tensor Loops

```c
__global__ void g_0001(const int N, double** TDm00,
    const double** TDm01, const double** TDm02) {
    const int a = threadIdx.y;
    const int b = blockIdx.y;
    const int x = blockIdx.x*blockDim.x + threadIdx.x;
    if ((x<N)&&(a<4)&&(b<4))
        TDm00[a+4*b][x]=TDm01[a+4*0][x]*TDm02[0+4*b][x] +
                     TDm01[a+4*1][x]*TDm02[1+4*b][x] +
                     TDm01[a+4*2][x]*TDm02[2+4*b][x] +
                     TDm01[a+4*3][x]*TDm02[3+4*b][x];
}
```

In the real code we use `__restrict__` flags on the pointer arguments for performance reasons. On the GPU, it is advantageous to store the tensor indices in a single, flattened array, since pointer indirections are relatively expensive. Similarly, unrolling expressions which on the CPU would have appeared as for loops prevents unnecessary serialization.

The variables `threadIdx.y`, `blockIdx.y`, etc, are used by CUDA to manage parallel data access. In CUDA, computations are abstracted as a three-dimensional grid of blocks, in turn composed of threads. Each thread represents a discrete computational process that will execute the instructions in the kernel. While differing threads issue the same instructions, they will normally do so upon differing data, since they may address memory using their unique block and thread indices (`threadIdx.y`, etc.). for loops are generally replaced with if statements such as that in Listing 4.20, which ensures that no thread make an out-of-bounds array access.

Physically, GPU resources are divided into streaming multiprocessors (SMs) composed of tightly coupled processing cores. Cores in a given SM execute instructions in lockstep, and share certain memory resources with one another besides the global memory accessible to the entire GPU. Dividing threads into blocks, which are always local to a particular SM, enables such resources to be safely utilized. Although we make no use of such resources, the blocksize is nevertheless important for us, since a single SM may operate upon only a certain number of blocks at one time. The SM hides latency by switching between those blocks when one stalls (for example because of a data dependency). A poor blocksize choice can result in low “occupancy”, i.e. the percentage of available blocks that are in fact simultaneously scheduled by a kernel. Low occupancies can impair latency hiding, since they can potentially result in a situation where an SM could remain active by switching blocks, but there are no blocks for it to switch to.

For this and other reasons, it is important to appropriately “tune” the kernel launch,
via choice of the number of blocks in the grid \((n_{\text{blocks}})\), and the number of threads in each block \((\text{blocksize})\). Those arguments are, in the case of Listing 4.20, fixed by the corresponding “wrapper” code presented in Listing 4.21.

Listing 4.21: CUDA wrapper corresponding to Listing 4.19.

```c
void CUDAWrapper_g_0001(const int N, double** TDm00, const double** TDm01, const double** TDm02){
    const int blocksize_x = 64;
    const int nblocks_x = sz / blocksize_x + (sz%blocksize_x == 0?0:1);
    const int blocksize_y = 4;
    const int nblocks_y = 4;
    const int blocksize_z = 1;
    const int nblocks_z = 1;
    const dim3 blocksize(blocksize_x, blocksize_y, blocksize_z);
    const dim3 nblocks(nblocks_x, nblocks_y, nblocks_z);
    g_0001<<<nblocks,blocksize>>>(sz, TDm00, TDm01, TDm02);
}
```

We expose the parallelism of the \(N\) data-independent spatial gridpoints by devoting the entire logical \(x\) dimension of the CUDA grid to them. We then use the four remaining thread addresses to parallelize over the indices of the left-hand side tensor, since the corresponding arrays are also data-independent. In principle, we could implement the Sum operator as a parallel reduction as well. However, this dramatically complicates automatic code generation and offers no advantage at dimensions 3 or 4. Therefore, we instead perform sums in serial, as demonstrated in Listing 4.20.

There are two cases in which we cannot parallelize over all the LHS components. The first is that of a symmetry. For example, Listing 4.15, which has a symmetry between the \(a\) and \(b\) indices, generates the kernel in Listing 4.22.

Listing 4.22: CUDA kernel corresponding to Listing 4.15.

```c
__global__ void g_0001(const int N, double** TDm00, const double** TDm01, const double** TDm02){
    const int b = threadIdx.y;
    const int x = blockIdx.x*blockDim.x + threadIdx.x;
    if ((x<N)&&(b<4)){
        for (int a=b; a<4; ++a){ /*
            TDm00[a+4*b][x]=TDm01[a+4*0][x]*TDm02[0+4*b][x] +
            TDm01[a+4*1][x]*TDm02[1+4*b][x] +
            TDm02[a+4*2][x]*TDm02[2+4*b][x] +
            TDm02[a+4*3][x]*TDm02[3+4*b][x] +
        */
```
TDm01[ a + 4 \times 2][x] \times \text{TDm02}[2 + 4 \times b][x] + \\
\text{TDm01}[ a + 4 \times 3][x] \times \text{TDm02}[3 + 4 \times b][x];

Because the number of passes through the for loop in marked by //∗ in Listing 4.22 depends on the value of b, parallelization of that loop would require different CUDA threads within a block to execute different instructions. Since all the threads in a particular SM share the same control circuitry, however, this is not possible. CUDA deals with this by having SMs that encounter so-called “divergent execution paths” run each one in serial. We avoid this by serializing explicitly.

The second case we cannot parallelize is the unusual one of an LHS tensor of rank greater than four. This exhausts the number of independent CUDA thread addresses, and so we must serialize the extra indices.

We tune our kernels using the following simple rules, designed to achieve maximum or high occupancy on all the GPUs in Table 3.1. We use (compare Listing 4.21) blocksize_x and nblocks_x to parallelize across the spatial grid, which leaves us four independent parameters with which to parallelize across LHS tensor indices. Each one will be used to parallelize a different index, and will thus be set to either 1, 3, or 4 (either no index, or the dimension of the relevant index). Therefore, blocksize_y \times blocksize_z will be either 1, 3, 4, 9, 12, or 16.

We now must set blocksize_x in order to control the total blocksize. This must be a multiple of 32, or else cores will be left idle, since GPU instructions are issued to groups of 32 in lockstep. An optimal total blocksize, allowing each SM to fully utilize its compute resources, will be one of a few values that depend on the particular GPU in question. On the M2090, for example, these are 192, 256, 384, 512, and 768. 256 and 512, in particular, achieve maximum occupancy across all three cards. These values can be achieved exactly when blocksize_y \times blocksize_z is 1, 4 or 16, in which cases we respectively set blocksize_x to 256, 64, or 32. Otherwise, we set blocksize_x to 64 (for blocksize_y \times blocksize_z = 3) or 16 (for 12), which are near-optimal. Note that this algorithm limits the maximum N that our code can handle to 65535 \times \text{blocksize_x}, which is always in the millions. This limit could be easily removed by for example serializing over extremely large grids, but this has not been necessary for our purposes.

Each SM has a single “register file” of extremely fast RAM used to store variables allocated within a kernel. During kernel launch, those registers are logically allocated to individual threads as necessary. If a kernel’s register demands are such that running all
possible threads would exhaust the register file, CUDA will restrict the number of blocks
assigned to each SM to compensate, thus lowering occupancy and possibly affecting the
above calculations. In the above calculations we assume this does not happen. Our
benchmarks (c.f. Table 4.1) show this assumption is usually, but not always, borne out.

The synchronization of Tensors presents an additional complication for CUDA which
is not present on the CPU. Recall that the individual arrays representing components
of a Tensor are not contiguous on the CPU, since that class does not assume those
components have identical memory footprints, and in fact permits modification of those
footprints after construction, for example by reshaping the component DataMeshes. The
CPU Tensors instead maintains an array of pointers, each addressing a particular tensor
element.

This design choice is sufficiently inextricable from SpEC that we must work around
it. But the obvious solution of maintaining an equivalent array of pointers on the GPU
has a significant performance impact if handled naively. The pointer array can change
after construction, so we cannot simply create a GPU equivalent once and assume it will
be always correct. On the other hand, the high latency of GPU array allocation and
synchronization makes copying a fresh array with each kernel launch unacceptable.

Instead, each Tensor is paired with a set of “GPUPointers” that store a copy of the
CPU pointer array, the GPU pointer array, and a reference to the relevant Tensor. When
the GPU pointer array is retrieved, we first ensure that the copies CPU array is identical
with that actually present in the Tensor, synchronizing only if it is not. This keeps the
number of necessary synchronizations to their bare minimum. If the GPU array is never
retrieved we do not create it at all, so that extra overhead is not incurred if a Tensor
never encounters a TLoops kernel.

4.5 Benchmarks

4.5.1 Methodology

We now turn to benchmarks of both the automatically-generated code and the expression-
template implementation. Due to the wide range of potential expressions, hardware,
compilers, and compiler options, it is not possible to do this fully comprehensively, but
we aim here to give a broad picture of our code’s behaviour.

Our kernels make no explicit attempt to reuse data once loaded, and we therefore
expect them to be bandwidth-bound or latency-bound (i.e. the limiting factor to their
performance is either the memory bandwidth of the device or the latency between succes-
A useful performance metric in this case is the “effective bandwidth” $	ext{BW}_{\text{eff}}$, which is the ratio between the amount of information which must be read and written by an operation with the time $t$ taken to execute that operation in practice. We measure $\text{BW}_{\text{eff}}$ in GB/s. $\text{BW}_{\text{eff}}$ will be maximal for a kernel which simply copies data, and decrease as operations spend significant time on computations or extraneous memory operations. Calling $N$ the spatial gridsize, $N_\epsilon$ the total number of tensor elements involved in the operation and $N_d$ the number of isolated doubles, we have

$$\text{BW}_{\text{eff}} = 8\text{bytes} \frac{N_\epsilon N + N_d}{t}. \quad (4.9)$$

For example, the expression in Listing 4.15 has $N_d = 0$ and $N_\epsilon = 42$ (16 elements each from $A$ and $B$, but only 10 from $C$ due to the symmetry), while the one in Listing 4.17 has $N_d = 0$ and $N_\epsilon = 19$ (3 from $D$, 4 from $F$, and 12 from $E$).

We begin with basic operations typical in relativity. We benchmark each such operation using three-dimensional tensor indices, at three levels of complexity. These operations are assignments

$$A_i = B_i, \quad (4.10)$$
$$A_{ij} = B_{ij}, \quad (4.11)$$
$$A_{ijk} = B_{ijk}, \quad (4.12)$$

additions,

$$A_i = B_i + C_i, \quad (4.13)$$
$$A_i = B_i + C_i + D_i, \quad (4.14)$$
$$A_i = B_i + C_i + D_i + E_i, \quad (4.15)$$

outer products,

$$A_{ij} = B_i C_j, \quad (4.16)$$
$$A_{ijk} = B_i C_j D_k, \quad (4.17)$$
$$A_{ijkl} = B_i C_j D_k E_l, \quad (4.18)$$
and contractions

\[ A_{ijkl} = B^m_{i} E_{mjkl}, \]  
\[ A_{ijkl} = C^n_j B^m_i E_{mnkl}, \]  
\[ A_{ijkl} = D^n_o C^n_j B^m_i E_{mno}. \]  

We furthermore benchmark two practical expressions that actually occur in numerical relativity, the first involving the extrinsic curvature \( K_{ij} \),

\[ K_{ij} = 2\alpha g_{ij} + \beta_i \beta_j, \]  

and the second (Eq. (4.6)) computing the spatial Christoffel symbols of the second kind \( \Gamma^i_{jk} \).

Finally, we will present under the label “GH” our TLoops port of the actual SpEC module which solves the generalized harmonic equations in the eponymous formulation of relativity theory. Roughly speaking, this code computes the second time-derivative of the spacetime metric \( \psi_{ab} \), as a function of first and second spatial derivatives. As such, the input data is primarily the \( \psi_{ab} \), its spatial derivatives \( \partial_i \psi_{ab} \) and \( \partial_i \partial_j \psi_{ab} \), and its first time-derivative \( \partial_t \psi_{ab} \). In total there are 22 input arrays. The equations involved consist of 25 separate TLoops expressions, with as many as four LHS indices, and as many as two contractions on the RHS. Overall, we estimate 381 spatial-gridsize arrays in the numerator of Eq. (4.9) for the GH operation, so that GH is about 4-100 times more bandwidth-intensive than the other benchmarked expressions. This reflects in the raw execution times: GH takes about 10 times longer to execute than the other benchmarks, though the overall execution time is normalized away by plotting \( BW_{eff} \).

TLoops can also handle transcendental functions and many other unary functions. Such functions occur often enough that automatic code-generation is warranted. But they only use a marginal fraction of overall runtime, and so we do not benchmark such functions at present.

We benchmark each expression on the following combination of hardware and code-path:

1. Automatically generated CUDA code executing on the three NVIDIA GPUs in Table 3.1, namely M2090, K80, and P100.

2. Automatically generated C code executing on the host processor (labeled ‘Accel-CPU’).
3. The TLoops expression templates executing on the host processor (labeled ‘NonAccel’).

4. The original SpEC code without TLoops code-simplifications, as an overall baseline (labeled SpEC).

The CPU code was compiled using gcc 4.8.1 using the -03, -fPIC, and -std=c++11 compiler flags. We also took benchmarks using intel 15.0.2 and the compiler flags -03 -xHost -fPIC -fp-model precise -std=c++11. The Intel code usually gives comparable or worse results to gcc (c.f. Figure 4.4), and so for visual simplicity only the gcc results are displayed in Figures 4.2, 4.3, and 4.5. The CPU timings were performed on a single core of an Intel Xeon E5-2620 CPU, which has a clock frequency of 2.0 GHz, a theoretical bandwidth of 42.7 GB/s, and a theoretical double-precision processing power of 8.0 GFLOP/s. We sampled gridsizes at multiples of 32 with decreasing resolution at increasing gridsize.

We have not made a systematic study of CPUs and compiler options, and do not intend for these results to reflect the potential CPU performance of our code. Improved results could almost certainly be achieved using a CPU with a higher clock frequency and e.g. vectorized instructions over multiple cores. This machine and these compiler options are, however, representative of realistic conditions under which SpEC might presently run. In particular, limitations to parallelism imposed by SpEC’s implementation of multdomain pseudospectral methods restrict it to a single CPU core per expression.

The M2090’s host processor is the same as used for the CPU test. We compiled the CPU code in this case using the Intel compiler with the same options as above. The K80 and P100 are hosted by somewhat faster POWER8 processors, and the code in these cases was compiled using xlc 13.1.4 with the flags -fPIC, -03, -std=c++11. The CPU’s performance should not be relevant to the GPU tests. We compiled the GPU code with CUDA 6.5 on the M2090 using -arch=sm_20. On the K80 (-arch=sm_37) and P100 (-arch=sm_60) we used CUDA 8.0.

### 4.5.2 Benchmarks of simple expressions

We are now ready to present benchmarks for the expressions corresponding to Eqs. (4.10)-(4.18) on each of the various hardware and code-path combinations. We execute each benchmark 21 times, discard the first, and take the median.

The results are summarized in Figure 4.2. Each panel of that figure corresponds to one particular expression, with the x-axis indicated grid-size, and the y-axis indicat-
Chapter 4. Tensor Loops

ing performance. We express performances in terms of effective bandwidth \( BW_{\text{eff}} \), c.f. discussion in Section 4.5.1 above.

Let us now discuss and interpret Figure 4.2. Focusing first on the GPUs, ‘M2090’, ‘K80’, and ‘P100’ all show the same overall trends. Execution time is roughly constant in the gridsize until occupancy is saturated, after which point it increases linearly. Since the number of memory transactions increases linearly in gridsize throughout, \( BW_{\text{eff}} \) shows linear increase up to the point of saturation, after which point it is constant. Since all the GPUs have essentially the same single-thread performance, they perform essentially identically until their respective points of saturation. However, newer cards (especially the P100) can support more parallel threads, resulting in a later point of saturation with a higher \( BW_{\text{eff}} \).

The saturation gridsize is most importantly determined by the left-hand side tensor rank: higher rank tensors saturate earlier. For example, in Figure 4.2 the saturation gridsizes are almost identical for \( A_{ij} = B_{ij} \) compared with \( A_{ij} = B_{i} C_{j} \), for \( A_{ijk} = B_{ijk} \) compared with \( A_{ijk} = B_{i} C_{j} D_{k} \), and for \( A_{i} = B_{j} \) compared with any of the addition operations Eqs. (4.13)-(4.15). This reflects our code’s parallelism over tensor indices, which is crucial for achieving good performance at gridsizes on the order of \( 10^4 \). The pattern would not persist past rank 4 or for symmetric indices, since we serialize in these cases. Post-saturation performance is mostly independent of the operation and usually quite close - slightly beneath a factor of 2 in the worst case of \( A_{ijkl} = B_{i} C_{j} D_{k} E_{l} \) - to the measured bandwidths from \texttt{bandwidthTest}.

In contrast, the three CPU execution-paths do not suffer from high latency, and so the \( BW_{\text{eff}} \) curves are generally quite flat with respect to gridsize. Nevertheless, several patterns are visible in the relative execution speed of the three CPU execution paths. Except sometimes for assignments (Eqs. (4.10)-(4.12)) at very large gridsize, the expression-template code (\texttt{NonAccel}) usually gives worse performance than either the automatically-generated C code (\texttt{AccelCPU}) or that \texttt{SpEC} without any \texttt{TLoops} simplifications (\texttt{SpEC}) by a factor of between about 3-10. These results are roughly in line with those obtained from Walter Landry’s FTensor [2,185], which is similar to \texttt{TLoops} running in \texttt{NonAccel} mode, and is presumably due to the compiler being less able to optimize the various templated expression templates.

More unexpectedly, \texttt{SpEC} and \texttt{AccelCPU} do not perform identically. While performance is usually comparable, \texttt{SpEC} is sometimes noticeably superior, especially for less complex operations at small gridsizes. \texttt{AccelCPU} differs from \texttt{SpEC} in two ways. First, the for loops over tensor indices appear directly in source code using \texttt{SpEC}, whereas \texttt{AccelCPU} routes through a few extra classes before reaching them. While we consider it unlikely,
Figure 4.2: TLoops performance benchmarks of assignment (top row), addition (middle), and outer products (bottom). Operations increase in complexity as panels move from left to right. Each panel shows the effective bandwidth of the automatically generated GPU code (labelled by the GPU used for the benchmarks), automatically generated CPU code (AccelCPU), and expression templates without automatic code generation (NonAccel). The black line (SpEC) shows the performance of SpEC without the use of the TLoops package.
the impaired performance for less complex operations may be due to some extra overhead from this routing. Second, SpEC handles the loop over gridpoints using expression templates, while AccelCPU uses an additional for loop. This may result in differing behaviour regarding e.g. the creation of temporaries and the use of cached memory in the machine code.

Now comparing the performance of the GPUs with the CPU executions paths, we note that the CPU generally gives better performance at small gridsize, but is eventually surpassed by the GPU. This is the expected behaviour: the CPU has superior single-thread performance, but the GPU has more capacity for parallelism. Compiler optimizations available to the CPU will likely also result in more efficient reuse of memory than on the GPU. This will make operations on the CPU less complex, but also make the floating-point performance of the hardware more relevant. Thus, the CPU has less of an advantage at small gridsize for more computationally intensive operations.

4.5.3 Benchmarks of more complex expressions

Let us now turn to the more complex operations corresponding to Eqs. (4.19)-(4.22), Eq. (4.6), and the GH equations, each described in Section 4.5.1. We proceed here as in Section 4.5.2, benchmarking six hardware and code-execution-path combinations as a function of gridsize, with results presented in Figure 4.3.

Figure 4.3 shows broadly similar features throughout: GPU performance increases linearly up to a saturation point and then is constant, while CPU performance is nearly flat. In particular, the $K_{ij}$ operation (Eq. (4.22), lower left panel of Figure 4.3) behaves essentially identically to Eq. (4.17) (lower left panel of Figure 4.2), to which it is indeed very similar in form.

The contraction operations (Eqs. (4.19), (4.20), and (4.21)) in the top row of Figure 4.3 show some new behaviour. On the CPU, we first of all notice that performance, while still independent of gridsize, worsens sharply as we move from left to right between panels. These operations are more strongly compute-bound than those discussed until now, although the form of our automatically generated code does not expose this. Since the CPU performs memory operations relatively better than floating-point computations, its performance degrades for operations involving more of the latter.

We also notice that AccelCPU gives better performance than does SpEC for these operations, which is the opposite behaviour as observed previously. We can only guess at the reason for this. Perhaps there is more opportunity for compiler optimizations for operations involving more floating-point operations, but the expression-templates over
Chapter 4. Tensor Loops

Figure 4.3: TLoops performance benchmarks of contraction (top row) and of practical numerical relativity operations (bottom). Each panel is formatted in the same way as in Figure 4.2. The leftmost and central operations in the bottom panel correspond respectively to Equations 4.22 and 4.6. The rightmost operation, “GH”, shows the performance of the entire SpEC module that advances the Einstein equations (in their generalized harmonic formulation) by a timestep.

Turning attention to the GPU curves, we see that the low-gridsize performance is almost exactly identical between panels. Due to the massive parallelism it must support, the CUDA compiler cannot make nearly so aggressive optimizations as can a modern C++ compiler, and so the automatically-generated code presumably behaves in the bandwidth-bound manner in which it is written. The saturation gridsize, however, does change, even though the number of LHS indices remains constant. Similarly, the post-saturation performance gets lower as we move from left to right.

This stems from the fact that the SpEC class Tensor is a list of arrays (one array over the spatial grid per tensor index), rather than a single contiguous one. Since the tensors
### Table 4.1: Per-thread register count (Regs) and theoretical occupancy (% Occ) for benchmarked TLoops operations on each GPU as measured by the NVIDIA visual profiler.

On the M2090, K80, and P100, register use begins to impair occupancy respectively at counts exceeding 21, 64, and 32. The GH operation is not profiled here since it does not consist of a single kernel.

<table>
<thead>
<tr>
<th>Operation</th>
<th>M2090 Regs</th>
<th>M2090 % Occ</th>
<th>K80 Regs</th>
<th>K80 % Occ</th>
<th>P100 Regs</th>
<th>P100 % Occ</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4.10)</td>
<td>10</td>
<td>100</td>
<td>10</td>
<td>93.8</td>
<td>12</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.11)</td>
<td>10</td>
<td>100</td>
<td>10</td>
<td>93.8</td>
<td>12</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.12)</td>
<td>10</td>
<td>83.3</td>
<td>10</td>
<td>93.8</td>
<td>12</td>
<td>93.3</td>
</tr>
<tr>
<td>(4.13)</td>
<td>14</td>
<td>100</td>
<td>14</td>
<td>93.8</td>
<td>14</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.14)</td>
<td>18</td>
<td>100</td>
<td>16</td>
<td>93.8</td>
<td>15</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.15)</td>
<td>20</td>
<td>100</td>
<td>21</td>
<td>93.8</td>
<td>18</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.16)</td>
<td>14</td>
<td>100</td>
<td>14</td>
<td>93.8</td>
<td>14</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.17)</td>
<td>18</td>
<td>83.3</td>
<td>16</td>
<td>93.8</td>
<td>16</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.18)</td>
<td>21</td>
<td>83.3</td>
<td>21</td>
<td>93.8</td>
<td>18</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.19)</td>
<td>28</td>
<td>72.9</td>
<td>29</td>
<td>93.8</td>
<td>24</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.20)</td>
<td>38</td>
<td>41.7</td>
<td>42</td>
<td>93.8</td>
<td>32</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.21)</td>
<td>50</td>
<td>41.7</td>
<td>56</td>
<td>93.8</td>
<td>48</td>
<td>62.5</td>
</tr>
<tr>
<td>(4.22)</td>
<td>21</td>
<td>87.5</td>
<td>42</td>
<td>93.8</td>
<td>32</td>
<td>93.8</td>
</tr>
<tr>
<td>(4.6)</td>
<td>34</td>
<td>62.5</td>
<td>32</td>
<td>93.8</td>
<td>32</td>
<td>93.8</td>
</tr>
</tbody>
</table>

are not contiguous each memory access actually involves two pointer indirections, one each to retrieve the appropriate component array and spatial gridpoint. For example an instruction such as \( \mathbf{d} = \mathbf{A}[i][j][x] \) must first load \( \mathbf{A}[i] \) from global memory, then \( \mathbf{A}[i][j] \), then finally \( \mathbf{A}[i][j][x] \). Since the first two loads are not in principle necessary, we do not include them in the numerator of \( \text{BW}_{\text{eff}} \). Since that numerator therefore underestimates the true number of memory transactions our computed \( \text{BW}_{\text{eff}} \) will be correspondingly lower.

The extra indirections also result in additional thread latency, since the thread must stall between the successive loads, and since the large number of loads may exhaust the SMs memory pipeline. This last effect could in principle be alleviated by staggering loads to avoid memory dependency, but this would complicate automatic code generation considerably. Future improvements will instead focus on making tensors contiguous.

The extra pointers finally result in extra thread-local memory allocations, increasing the kernel’s per-thread register count. Each streaming multiprocessor (SM) in a GPU has physically a single register file that is logically allocated to threads as needed. Each SM is also theoretically capable of simultaneously executing a certain number of warps,
Chapter 4. Tensor Loops

100
each consisting of 32 threads, but only if the per-thread register count is small enough that these warps do not collectively exhaust the register file.

On the M2090, K80, and P100 respectively, this occurs when the per-thread register count exceeds 21, 64, and 32. The SMs on the K80 and P100 have equally sized register files (of 256kb, compared to 128kb on the M2090), but the P100 SMs can also potentially execute more warps, resulting in a lower register threshold for maximum occupancy. If the limit is saturated by a large threshold, occupancy will significantly decrease, resulting in an earlier point of saturation with worse asymptotic performance. We never exceed this threshold on the K80 (Table 4.1) but it does sometimes become relevant for operations involving contractions.

Finally, let us turn attention to the GH operation, in the lower right panel of Figure 4.3. On the GPU, the transition from linear to constant performance growth is not nearly so sharp as for the single-expression operations. This presumably reflects an averaging out between the many saturation points of the differing expressions within GH. GH also displays (in all cases) noticeably worse performance compared to its predecessors in this discussion. The GH operation consists of many successive individual kernels, many of which are complex contractions; thus, the above discussion of contractions applies here as well. On top of this, the many individual kernel launches add latency to the GPU execution time.

4.5.4 Impact of CPU compiler

In Figure 4.4, we show some benchmarks illustrating the relative performance of gcc vs. Intel compilers operating upon our code. Generically, but not always, gcc gives better performance. The difference is most stark for the C++11 expression templates of NonAccel, which work over an order of magnitude faster using gcc throughout. gcc also gets uniformly better performance out of the AccelCPU code, though the difference is less dramatic. Without TLoops ("SpEC"), the compilers do behave differently, but their relative performance varies between operations.

4.5.5 Impact of templating over Tensor-indices

From the perspective of automatic GPU-porting, an alternative approach to TLoops would be to automatically generate code from SpEC’s existing spatial-gridpoint expression templates. For example, in Listing 4.4, one might automatically generate code only for the interior operation, and not for the full expression including the for loops over i and j. This would be much simpler to write, and would require no source code modifications.
Chapter 4. Tensor Loops

\[ A_i = B_i + C_i \]

\[ A_i = B_i + C_i + D_i + E_i \]

Figure 4.4: TLoops performance of selected operations illustrating the respective performance of the CPU code when compiled using gcc (solid lines) vs Intel (dashed) compilers. The detailed version and compiler arguments are given in Section 4.5 of the text. Formatting is otherwise identical to Figures 4.2 and 4.3.

at all.

Chronologically, this approach was the first we tried. TLoops was motivated by its strongly negative performance impact on the SpEC code proper. The performance decrease accounting for this is illustrated in Figure 4.5. Here, we benchmark various expressions using the P100 GPU and the two TLoops CPU execution pathways. For the lines marked Tensors, TLoops is used to represent the full expression, as in Listing 4.2. For those marked Arrays, it is used only to represent operations over individual Tensor elements, which therefore are surrounded by explicit for loops in source code, as in Listing 4.1. While the respective performance of the two strategies is comparable on the CPU, on the GPU TLoops expressions are vastly superior, particularly at realistic gridsizes between about 1000 and 60000.

Templating over tensor indices is advantageous on the GPU for three reasons. First, launching a GPU kernel carries an overhead of about 20 µs. In the array loop approach this overhead needs to be paid once per every free and contracted index in the operation. Automatically ported operations will usually be small, and in practice launch overhead is very often the dominant expense. A TLoops operation, on the other hand, launches only one kernel. Second, TLoops operations are parallelized over the left-hand side tensor indices as well as the spatial grid, whereas the array loop approach can parallelize only over the spatial grid. In principle the array loop approach could achieve some index-
Figure 4.5: TLoops performance of selected operations showing the performance advantage attained by templating over entire tensors (solid lines, “Tensors” in the legend) rather than individual component arrays (dashed lines, “Arrays”). These plots are otherwise formatted in the same way as in Figures 4.2 and 4.3. To avoid visual confusion we display results from only one GPU (the P100), but the qualitative behaviour is the same for each.
level parallelism via concurrent execution of GPU kernels. However, the aforementioned launch overhead synchronizes the device, preventing concurrent execution in practice.

4.6 Conclusion

We have presented a software package, TLoops, which allows tensor-algebraic expressions to compile and execute in C++ code. TLoops can also automatically generate equivalent C++ or CUDA code to these expressions, which can be linked back to a second compilation. We have shown this automatically generated code to give identical or comparable performance compared to the code SpEC uses by default, and that the CUDA code often outperforms the CPU. Even at only moderate gridsizes of a few 1000, the CUDA code often comes close to the peak (memory-bound) performance of the GPU.

Significant opportunity remains for improvement. The code at present is intertwined with the rest of SpEC. We hope to separate it from the latter into an independent open-source library. Opportunity for performance improvements also exists. In particular, we are working on adopting a contiguous Tensor class within SpEC. This will allow for simpler, faster automatic code.

We hope the simplifications to coding effort made possible by TLoops may speed the development of future code, inside and outside of numerical relativity.
Chapter 5

Frequencies and Resonances of Generic BBH Inspirals

5.1 Abstract

Binary black holes which are both eccentric and undergo precession remain unexplored in numerical simulations. We present simulations of such systems which cover about 50 orbits at comparatively high mass ratios 5 and 7. The configurations correspond to the generic motion of a nonspinning body in a Kerr spacetime, and are chosen to study the transition from finite mass-ratio inspirals to point particle motion in Kerr. We develop techniques to extract analogs of the three fundamental frequencies of Kerr geodesics, compare our frequencies to those of Kerr, and show that the differences are consistent with self-force corrections entering at first order in mass ratio. This analysis also locates orbital resonances where the ratios of our frequencies take rational values. At the considered mass ratios, the binaries pass through resonances in one to two resonant cycles, and we find no discernible effects on the orbital evolution. We also compute the decay of eccentricity during the inspiral and find good agreement with the leading order post-Newtonian prediction.

5.2 Introduction

The recent landmark detections of gravitational waves by the Advanced LIGO interferometers gave definitive proof that binary black hole (BBH) systems exist and merge in nature [7–9]. Detection and characterization of the resulting signals relies on knowledge of the expected gravitational waveforms for tasks as varied as detection, parameter esti-
mation, and tests of general relativity [72]. The use of theoretical waveforms enhances gravitational wave (GW) astronomy’s scientific potential where waveform models are available, as demonstrated by the GW detections and analyses in Advanced LIGO’s first observing run [6]. Conversely, the absence of accurate and reliable waveform models may limit detection sensitivity, and hinders source parameter estimation and tests of gravitational theories.

Due to both physical motivations and computational complexity, waveform modeling for BBH systems has focused on quasi-circular binaries. Orbital eccentricity is damped away quickly during a GW-driven inspiral of two compact objects [240] because GW emission peaks at periastron. Thus “field binaries” formed from the respective collapse of both partners in a high-mass stellar binary are expected to be almost exactly circular by the time they enter the sensitive band of ground-based GW detectors. Quasi-circularity removes the two degrees of freedom related to orbital eccentricity, and thus reduces the dimensionality of the BBH parameter space that needs to be modelled. Mature waveform models exist for quasi-circular, aligned-spin BBH systems [175, 296], while models of quasi-circular BBH systems with generic spin (i.e. precessing binaries) are maturing quickly (e.g. [42, 143, 237]). These waveform models are based on direct numerical solutions of Einstein’s equations (e.g. [51]) that provide the gravitational waveforms for the late inspiral and merger of the two black holes; such numerical simulations, too, are most mature for quasi-circular systems (e.g. [17, 90, 157, 168, 171, 219]).

In contrast, very few numerical simulations have been performed for eccentric BBH systems [111, 130, 158, 159, 218, 287], and a systematic numerical exploration of the eccentric parameter space has not even started. Complete inspiral-merger-ringdown waveform models for eccentric BBH systems are also in a nascent state, with current models [163] neither reaching the accuracy of quasi-circular models, nor accounting for black hole spins (not even aligned-spin).

Several astrophysical scenarios have been explored in recent years which can lead to nonzero eccentricity late in the inspiral: Within dense stellar environments such as globular clusters (GCs) and galactic cores, binaries can form dynamically (e.g. [217, 258, 259, 263]), and there can be significant residual eccentricity from direct dynamical capture [162, 183, 250, 307]. The scattering of single stars off binaries [264] can also lead to high eccentricities, although these events are likely rare [37]. Even more promisingly, eccentricity can be generated by the Kozai-Lidov mechanism [116, 120, 121, 184, 195, 214, 223] in three body systems with moderate separations [35, 36, 58, 173, 275]. Such triples can occur in GCs [37, 323] or galactic nuclei [36], and may provide a source for eccentric compact binaries in advanced detectors [34, 37, 290] (in particular reference [37] estimates
$e > 0.1$ should occur at rate of about $0.2 \text{ yr}^{-1}$, but see also [29]).

Ultimately gravitational wave observations will be the primary tool to measure eccentricity of compact object inspirals and to constrain the rate of eccentric compact mergers. Unfortunately, the present detection template bank used by LIGO is entirely circular [6, 10], hindering detection of systems with in-band eccentricities above about $e \sim 0.1$ [98, 163, 164]. Due to this same absence of eccentric waveform models, parameter estimation must likewise assume quasi-circularity.

Looking ahead, the “extreme mass-ratio inspirals” (EMRIs) visible to space-based detectors such as LISA [28, 273] may be detected at quite large eccentricities (e.g. [30]). While eccentric waveforms are being developed [163, 295], further numerical and analytical modeling of eccentric systems are required to constrain eccentricity and prepare for future ground- and space-based detections of eccentric systems.

Eccentric systems also yield additional insight into strong-field dynamics, especially at high mass ratios [20, 186, 203]. While circular orbits must remain outside the separatrix [322] until the final plunge, eccentric orbits can reach as deep as the innermost bound orbit, thus probing stronger gravitational fields [48]. Furthermore, binaries which are both eccentric and precessing open up the possibility of qualitatively new phenomena when orbits pass through resonances and radiation reaction effects accumulate secularly [117, 118].

This paper presents a survey of 12 numerical simulations over a range of initial eccentricities as high as $e = 0.2$, mass ratios $q = 5$ and 7, and initial inclinations between the spin and the orbital plane between $0^\circ$ and $80^\circ$, performed with the Spectral Einstein Code (SpEC) [3]. Note these simulations were performed using the standard version of SpEC, and not the GPU port presented in Chapter 3. We focus on higher mass ratios in order to make contact with the extreme mass ratio limit, where the BBHs can be modelled to leading order by bounded motion of a test particle in a Kerr spacetime. Beyond the test particle limit, the motion is corrected by higher order, self-force (SF) effects due to the spacetime perturbations sourced by the particle [108]. In this case the ratio of the particle mass $\mu$ to that of the Kerr black hole $M_{\text{Kerr}}$ serves as a small expansion parameter. While SF results for Kerr are still in development [213, 276, 310, 312], mounting evidence suggests that SF results may be extended much closer to the nearly equal mass cases than expected [187–189], which is the regime appropriate for our our high (but not extreme) mass ratio binaries. Meanwhile, the lowest order approximation to the SF-expansion, namely the test particle limit, incorporates the effects of strongly curved spacetime, highly relativistic motion, and is fully understood. At the same time, post-Newtonian approximations have difficulty describing higher mass ratio binaries, where
the constituents remain at close separations for many orbits. In addition, higher mass ratio binaries can potentially remain at orbital resonances for multiple cycles of the resonance, and our simulations allow for the first investigation of the role of passage through resonances in BBHs.

In order to compare to the test particle limit and investigate these SF corrections, our primary interest is the computation of the characteristic frequencies in the azimuthal ($\Omega^\phi$), radial ($\Omega^r$), and polar ($\Omega^\theta$) directions. Except very close to the separatrix these frequencies furnish a one-to-one map with (self-force corrected) Kerr geodesics [322] and thus provide a natural point of comparison with the latter. In the Kerr spacetime the effect of timelike coordinate transformations that do not involve time-dependent rotations is to multiply all of the frequencies by the same factor [270]. Thus their ratios $K^{ab} \equiv \Omega^a / \Omega^b$ are insensitive to such transformations. In the particular case of nearly circular orbits, the periastron precession rate of BBHs simulations as encoded in $K^{r\phi}$ was investigated [188, 189], showing that analytic approximations provide highly accurate descriptions of the simulations. Additionally, the precession rates $K^{ab}$ are of interest because the points in the orbit when these frequency ratios are rational are precisely the moments of orbital resonance. Their extraction can be used to characterize and explore such resonances in our simulations.

Extracting the characteristic frequencies accurately from an eccentric, precessing numerical relativity simulation turns out to be challenging due to both strong dissipation as well as modulations arising from interactions of radial and polar motion. Fourier-based methods of frequency extraction require several orbits to achieve percent level accuracy, resulting in unacceptable dissipative contamination. We therefore rely upon time-domain methods based on intervals between successive periastron passages, which achieve better accuracy with only a single orbit.

This paper is structured as follows. In section 5.3 we introduce the basic context of our analysis by discussing generic test orbits in the Kerr spacetime. In section 5.4 we review the simulations we have performed in detail and describe the association we make between simulation trajectories and Kerr geodesics. In section 5.5 we detail our frequency extraction methodology and its results for our equatorial runs. Section 5.6 provides the same analysis for the inclined runs, and showcases our ability to detect resonances in these simulations. We discuss our results and future directions in section 5.7.
Chapter 5. Frequencies and Resonances of Generic BBH Inspiral

5.3 Motion in Kerr spacetimes

We interpret our high mass ratio, eccentric, and precessing simulations in terms of motion in the Kerr spacetime. From the perspective of dynamical systems, bound test orbits in Kerr form an integrable Hamiltonian system. Such systems admit action-angle coordinates, in which the generalized momenta are the conserved actions $J^a$. The conjugate positions are circulating angles which evolve at fixed frequencies. The motion of the particle is multiperiodic, and can be expanded in a Fourier series in the frequencies. In Hamiltonian perturbation theory the action-angle formalism elucidates the perturbed motion, the loss of integrability, and the onset of chaos [194].

For Kerr geodesics, the transformation to action-angle variables was first discussed by Schmidt [270]. The resulting $J^a$ are geometric invariants, and the associated proper time frequencies describe the motion in a coordinate-independent manner. The frequencies measured by alternative observers are related to the proper time frequencies through multiplication by a Lorentz-like factor. In particular, distant inertial observers measure frequencies $\Omega^a$.

Self-force corrections to geodesic motion decompose usefully into conservative corrections to the orbital dynamics and dissipative effects which drive inspiral [215]. The small size of the dissipation means that this system is amenable to evolution using a two-timescale approach [160, 245]. We work in the paradigm where (accounting for the slow dissipation) the fundamental frequencies are perturbed by the conservative SF effects. This perturbed Hamiltonian perspective on the SF problem has been used to compute invariant quantities in Kerr spacetimes [169]\(^1\). Our goal in sections 5.5 and 5.6 is to extract the fundamental frequencies from our simulations, which have physical meaning in terms of precession rates of the binary as viewed by distant, inertial observers. The differences between these frequencies and those of Kerr can be viewed as (possibly high-order) self-force corrections; thus our work may be useful for comparison with future self-force results.

5.3.1 Geodesic orbits in Kerr

A test particle on a bound orbit in the Kerr spacetime has four conserved quantities: the specific energy $E$, the specific angular momentum along the symmetry axis $L_z$, the specific Carter constant $Q$, and the Hamiltonian constraint $\mathcal{H} = (1/2)g^{\mu\nu}p_\mu p_\nu = -\mu^2/2$, where $\mu$

\(^1\)The formal validity of this perturbative approach depends on the underlying dynamics being Hamiltonian and integrable. The conservative self-force dynamics in Kerr away from resonance were shown to have these properties in [125].
denotes the mass of the test particle. It is convenient to use Boyer-Lindquist coordinates $x^\mu = (t, r, \theta, \phi)$ (see 5.8.1) and to define the Carter-Mino time $\lambda$ through $[79, 215]$

$$\frac{d\tau}{d\lambda} = \rho^2, \quad \rho^2 = r^2 + a^2 \cos^2 \theta,$$

(5.1)

where $\tau$ is the proper time of the trajectory. Parameterized by $\lambda$ rather than $\tau$, the equations for the radial and polar motions of the particle separate,

$$\left(\frac{dr}{d\lambda}\right)^2 = R(r), \quad \left(\frac{d\theta}{d\lambda}\right)^2 = \Theta(\theta),$$

(5.2)

where the potentials $R$ and $\Theta$ are given in (5.28) and (5.29). This results in independent cyclic motions in the $r$ and $\theta$ directions.

Unfortunately, the equations of motion for $t(\lambda)$ and $\phi(\lambda)$ depend on both the radial and polar motions,

$$\frac{dt}{d\lambda} = T_r(r) + T_\theta(\theta) + aE_z, \quad (5.3)$$

$$\frac{d\phi}{d\lambda} = \Phi_r(r) + \Phi_\theta(\theta) - aE,$$

(5.4)

with the potentials on the right hand side given by (5.30) and (5.31). This means that $r(t)$, $\theta(t)$, and $\phi(t)$ are multiperiodic, complicating the extraction of the fundamental frequencies of motion. For geodesics, this problem can be solved by moving to action-angle coordinates $[270]$.

When studying bound orbits it is useful to introduce a Keplerian parametrization of the orbit, which represents the radial motion as a precessing eccentric orbit. The radius evolves as

$$r = \frac{pM}{1 + e \cos \chi^r},$$

(5.5)

where $p$ and $e$ respectively denote the semilatus rectum and eccentricity of the orbit. The phase $\chi^r(\lambda)$ represents the position of the particle along the precessing ellipse. The particle oscillates between the periastron $r_p = p/(1 + e)$ and apastron $r_a = p/(1 - e)$. Similarly, one can express the polar motion as an oscillation between symmetric turning points above and below the equatorial plane, by writing $[106, 124]$

$$\cos \theta \equiv \cos \theta_{\text{min}} \cos \chi^\theta,$$

(5.6)

where $\theta_{\text{min}}$ is the smallest polar angle the particle reaches at the height of its vertical motion. We define the inclination of the orbit by $i = \pi/2 - \theta_{\text{min}}$ as the inclination angle.
of the orbit above the equatorial plane. Note that our use of $i$ to define the inclination angle of our orbit differs from the commonly used (e.g. [165]) inclination angle $\iota$ defined through the constants of motion by $\cos \iota = L_z/(Q + L_z^2)^{1/2}$.

Together, $(p, e, i)$ characterize bound orbits. Given these orbital parameters, we can compute the corresponding constants of motion $(E, L_z, Q)$ [270] and solve the equations of motion, see e.g. [124].

### 5.3.2 Fundamental frequencies of Kerr

In the case of equatorial, eccentric orbits, the fundamental frequencies are straightforward to define. The particle oscillates between periastron and apastron, while advancing in the azimuthal angle $\phi$. The two relevant frequencies are the frequency between successive radial passages, $\Omega^r$, and the azimuthal frequency averaged over successive passages, $\Omega^\phi$.

The frequency ratio $K^{r\phi} = \Omega^r/\Omega^\phi$ measures the periastron precession rate. It is a well-defined observable that can be measured by distant inertial observers, for whom the particular coordinate time that the frequencies reference (whether it be $t$, $\tau$, or $\lambda$) divides out.

For generic, non-equatorial orbits the situation is more complicated. In terms of $\lambda$ the radial and polar motions decouple. However, azimuthal motion is modulated by both the radial and polar motions. Similarly, the Boyer-Lindquist time $t(\lambda)$ along the world line is modulated by the radial and polar motions. This results in multiperiodic behaviour in the coordinate graphs: the time between successive coordinate extrema is variable, and only the long-term average time between extrema approaches the fundamental periods $T^a = 2\pi/\Omega^a$.

Thus the fundamental frequencies $\Omega^r$, $\Omega^\theta$, $\Omega^\phi$ are infinite time averages. Once again, the ratios of frequencies $K^{r\phi}$, $K^{r\theta} = \Omega^r/\Omega^\theta$, and $K^{\theta\phi} = \Omega^\theta/\Omega^\phi$ eliminate the dependence on the particular choice of time coordinate, and are related to the precession rates of the periastron and of the instantaneous orbital plane.

We use the method of Schmidt [270] to compute the fundamental frequencies in terms of our chosen time parameter. Fujita and Hikida [124] provide analytic solutions for the bound orbits and their fundamental frequencies in terms of elliptic integrals. We do not exploit these, instead numerically integrating the geodesic equations to validate our frequency extraction methods discussed in section 5.4. We choose to use frequencies $\Omega^a$ in terms of Boyer-Lindquist time $t$. The precise choice of time-coordinate will cancel in the frequency ratios, so long as the coordinates under study do not differ by time-dependent rotations.
5.3.3 Orbital resonances

Unlike the familiar quasi-circular inspirals, eccentric and especially eccentric, precessing systems can access the qualitatively new dynamical effect of orbital resonances. These occur when the fundamental frequencies $\Omega^a$ are in (or near) rational ratio. In that case the trajectory is exactly (or approximately) closed in phase space and perturbations can accumulate secularly. During resonance the perturbed system’s evolution can deviate dramatically from its unperturbed counterpart. In the generic case of a Hamiltonian system this leads to “islands” of chaotic motion in the phase space. When dissipation is included perturbed systems pass through resonances. This may lead to distinct changes in the evolution of the frequency ratios (see, e.g. [209]), the resonant “kicks” discussed below, or even capture into the resonance [152, 308]. If the passage is fast enough, however, the system may not display chaotic behaviour or other signatures of resonant passage. For late-inspiral BBH systems no evidence for such chaotic motion has been observed, presumably since the dissipative timescales are too short for any significantly ergodic motion to manifest.

Resonances can nevertheless have an important effect on the motion of high mass-ratio systems, due to concordant asymmetries in the gravitational radiation reaction. The emission of energy, momentum, and angular momentum to infinity is controlled by the Weyl scalar $\Psi_4$, which when sourced by bound orbits in Kerr can be Fourier expanded as [106]

$$\Psi_4 = \frac{1}{(r - ia \cos \theta)^4} \sum_{lmkn} R_{lmkn}(r) S_{lmkn}(\theta) e^{im\phi - \Omega_{mkn} t}, \quad (5.7)$$

where $R_{lmkn}(r)$ is the radial wave function which solves the sourced radial Teukolsky equation [299], $S_{lmkn}(\theta)$ are the spin-weight $s = -2$ spheroidal harmonics, and

$$\Omega_{mkn} = m\Omega^\phi + n\Omega^\theta + k\Omega^r. \quad (5.8)$$

Fluxes to infinity are built from the time integrals of $|\Psi_4|^2$, followed by angular integrals over the sphere at infinity, possibly weighted by additional angular terms. The result is interference between harmonics $(m, n, k)$ and $(m', n', k')$. Typically the interference oscillates rapidly and does not contribute to the time integral. For example, for the flux of energy only terms where $(m, n, k) = (m', n', k')$ contribute.

At resonances, however, the generically-unimportant terms accumulate, leading to resonantly enhanced or diminished fluxes [118]. Resonances can also lead to gravitational wave beaming: during resonance the BBH trajectory (or its projection into the $r$-$\theta$ plane) closes on itself, taking the form of a Lissajous Figure. That Figure need not be symmetric.
in space, and gravitational wave emission need not be symmetric either. These resonant kicks have been studied for circular, precessing orbits at $\theta-\phi$ resonances [161], and for equatorial, eccentric orbits at $r-\phi$ resonances [309]. In these studies, the resonances required for symmetry breaking of the orbit are high order and occur only for very close orbits. For example, the $r-\phi$ kicks require $K^{r\phi} = 1/p$ with $p$ an integer and $p \geq 2$; such orbits are zoom-whirl orbits. Initial investigation in [309] indicated that $r-\phi$ kicks are relatively strong and may be effective for inspirals with mass ratios $q = m_1/m_2 \gtrsim 5$. However, these kicks require that a finite mass ratio inspiral achieve these extreme orbits before plunge and merger.

Resonant kicks do not lead to chaotic motion since their influence is self-limiting: the dissipation will inevitably push the inspiral off the resonant trajectory. At this point the system will resume its approximately adiabatic motion. Nevertheless the precise kick dynamics depend sensitively on the orbital phase at which the binary enters resonance. This would challenge a description of systems undergoing resonant kicks using e.g. a template bank of gravitational waveforms.

Of particular interest are the resonances between radial and polar motion described in [53,118]. These resonances cause secular accumulation in SF effects and drive a rapid change in the conserved quantities, especially the Carter constant $Q$. In the extreme mass-ratio limit $\mu/M \to 0$ an $O(1)$ change to the phase prior to resonance leads to an $O(\sqrt{M/\mu})$ correction to the phase post-resonance. Importantly, these resonances are effective at many more resonant frequencies than the kicks. Any rational ratio for $\Omega'$ and $\Omega^\theta$ will do, although lower order resonances like 2:3 or 3:4 are expected to have a larger effect than higher orders. Systems passing through successive $r-\theta$ resonances could accumulate a sensitive dependence on initial conditions.

While resonances are a high mass-ratio effect, the precise value of $q$ at which they may become important is as yet unknown, partly due to the current absence of any systematic numerical studies of eccentric inclined black hole binaries. It is indeed possible that resonances may become relevant at the mass ratios accessible, or nearly accessible, to numerical relativity. This would potentially seriously frustrate attempts by LIGO to measure moderate mass-ratio eccentric inspirals using matched filtering. In sections 5.5 and 5.6 we test to see whether the high mass ratio, eccentric and generic inspirals we discuss in section 5.4 display interesting resonant behaviour.
5.4 Numerical relativity simulations

5.4.1 Simulations of eccentric, precessing black hole binaries

We perform simulations with the Spectral Einstein Code [3], which uses a generalized harmonic formulation [123, 128, 197, 249] to integrate the Einstein equations in damped harmonic gauge [86, 198, 292]. The code uses an adaptively refined grid [204] between two sets of boundaries. Lying within the holes, the “excision boundaries” are chosen to conform to the shapes of the apparent horizons [150, 210, 233, 292]. After merger there is only one excision boundary [150, 210]. Being inside the holes, the excision boundaries are pure-outflow and no boundary conditions are required. The grid extends from the excision boundaries to an artificial outer boundary endowed with constraint-preserving boundary conditions [197, 256, 257]. The evolution proceeds from the construction [241] of quasi-equilibrium [81, 206] initial conditions satisfying the Einstein constraint equations [325].

We run each of our simulations at three resolutions, which we label L1 (lowest resolution), L2, and L3 (highest resolution). Each choice of resolution provides different tolerances on the adaptive mesh refinement. Each resolution thus has a different refinement history. While this prevents us from showing strict convergence of quantities extracted from the simulations, the range quantities take over the three resolutions is a measure of our numerical error. When it is practical we plot all three resolutions, and when we report a single result from a given simulation it is from the highest resolution, L3. All of our results are consistent across resolutions.

We label the masses of our black holes $m_1$ and $m_2$, using the convention that $m_1 > m_2$. The total mass of the black holes $M = m_1 + m_2$ sets all scales in our simulations. Our black holes have angular momenta $S_1$ and $S_2$, computed using quasi-local angular momentum diagnostics [206, 235], and we define dimensionless spin vectors through $\chi_i = S_i / m_i^2$, where here $i = 1, 2$ labels the black holes. We focus on two sequences of simulations. One sequence has mass ratio $q = m_1 / m_2 = 5$ and $\chi_1 = |\chi_1| = 0.6$ and the other sequence has $q = 7$ with $\chi_1 = 0.8$. In both cases, we set $\chi_2 = 0$. These parameters were chosen so that the binary orbits might be modelled by motion in Kerr spacetime with mass $m_1$ and spin parameter $\chi_1$, together with radiation reaction effects and finite mass ratio corrections to the motion. By selecting two choices of BH parameters we have some freedom to investigate the effect of varying those parameters while keeping computational expense manageable.

For our two choices of $(q, \chi_1)$ we ran two sets of simulations, using “low” and “high”

\(^2\)Note that for coordinate simulation quantities such as $\chi$ we use a flat Euclidean norm.
Figure 5.1: Coordinate trajectories for two $q = 7$, $\chi_1 = 0.8$, high eccentricity simulations. Left: An equatorial inspiral, depicted up to the final orbit before merger. Right: About 28 periastron passages from the middle of an inspiral with a $40^\circ$ inclination. Both panels show a 3-dimensional perspective view.

initial eccentricities. We targeted the initial eccentricities by using simple Keplerian relations between the initial orbital separation and angular velocity of the binary at the moment of apastron passage, where we began our simulations. Specifically, the initial data solver takes as input an initial expansion factor $\dot{a}_0$, an initial orbital frequency $\Omega_0$, and an initial coordinate separation distance $D_0$. We set $\dot{a}_0$ to zero in all cases, to fix the orbit at apastron. For a given initial distance $D_0$, we target a Newtonian eccentricity $e_N$, using the so-called “vis-viva” equation (which expresses energy conservation for the orbit) at apastron,

$$\Omega_0^2 = (m_1 + m_2) \left( \frac{1 - e_N}{D_0^3} \right), $$(5.9)

to solve for the appropriate $\Omega_0$. We chose $e_N = 0.2$ for our “low eccentricity” and $e_N = 0.3$ for our “high” eccentricity runs.

The initial separation $D_0$, together with the achieved eccentricity $e$ and the mass ratio $q$, controls the length of the inspiral. In order to facilitate accurate frequency extraction we set the simulations to be quite long, using initial distances $D_0$ of 19.5$M$ and 21.125$M$. This gave the same initial Newtonian semi-major axis of $D_0/(1 + e_N) = 16.25M$ for all of our simulations, and they proceeded through $\approx 30 - 60$ radial oscillations before merger.

To investigate the effects of precession we ran each simulations for each of the above choices of $(q, \chi_1)$ and $e_N$ three times, initializing the spin vector $\chi_1$ to form angles of $0^\circ$, $10^\circ$, and $20^\circ$ with the computational $z$-axis in the $q = 5$, $\chi_1 = 0.6$ cases and $0^\circ$, $40^\circ$, and $80^\circ$ in the $q = 7$, $\chi_1 = 0.8$ cases. The $z$-axis is normal to the initial orbital plane. These choices of initial inclinations of the orbital plane to the spin of the more massive hole
Table 5.1: Simulations used in this study. Tabulated here are mass-ratio $q$, dimensionless spin $\chi_1$ of the larger black hole, initial separation $D_0$, initial orbital frequency $\Omega_0$, the Newtonian eccentricity $e_N$ used to choose $\Omega_0$, the actual eccentricity of the simulation at its start $e_0$, and the initial inclination $i$. The last two columns report the number of azimuthal and radial cycles during the inspiral.

<table>
<thead>
<tr>
<th>Run</th>
<th>$q$</th>
<th>$\chi_1$</th>
<th>$D_0$</th>
<th>$\Omega_0 \times 10^2$</th>
<th>$e_N$</th>
<th>$e_0$</th>
<th>$i$</th>
<th>$N_\phi$</th>
<th>$N_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>q5i00_low-e 5</td>
<td>5</td>
<td>(0, 0, 0.6)</td>
<td>19.5</td>
<td>1.0387</td>
<td>0.2</td>
<td>0.07689</td>
<td>0°</td>
<td>55</td>
<td>41</td>
</tr>
<tr>
<td>q5i10_low-e 5</td>
<td>5</td>
<td>(-0.10419, 0, 0.59089)</td>
<td>19.5</td>
<td>1.0387</td>
<td>0.2</td>
<td>0.07700</td>
<td>10°</td>
<td>55</td>
<td>42</td>
</tr>
<tr>
<td>q5i20_low-e 5</td>
<td>5</td>
<td>(-0.20521, 0, 0.56382)</td>
<td>19.5</td>
<td>1.0387</td>
<td>0.2</td>
<td>0.07749</td>
<td>20°</td>
<td>54</td>
<td>41</td>
</tr>
<tr>
<td>q5i00_high-e 5</td>
<td>5</td>
<td>(0, 0, 0.6)</td>
<td>21.125</td>
<td>0.8617</td>
<td>0.3</td>
<td>0.21260</td>
<td>0°</td>
<td>42</td>
<td>31</td>
</tr>
<tr>
<td>q5i10_high-e 5</td>
<td>5</td>
<td>(-0.10419, 0, 0.59089)</td>
<td>21.125</td>
<td>0.8617</td>
<td>0.3</td>
<td>0.21280</td>
<td>10°</td>
<td>42</td>
<td>31</td>
</tr>
<tr>
<td>q5i20_high-e 5</td>
<td>5</td>
<td>(-0.20521, 0, 0.56382)</td>
<td>21.125</td>
<td>0.8617</td>
<td>0.3</td>
<td>0.21340</td>
<td>20°</td>
<td>41</td>
<td>31</td>
</tr>
<tr>
<td>q7i00_low-e 7</td>
<td>7</td>
<td>(0, 0, 0.8)</td>
<td>19.5</td>
<td>1.0387</td>
<td>0.2</td>
<td>0.06657</td>
<td>0°</td>
<td>74</td>
<td>59</td>
</tr>
<tr>
<td>q7i40_low-e 7</td>
<td>7</td>
<td>(-0.51423, 0, 0.612836)</td>
<td>19.5</td>
<td>1.0387</td>
<td>0.2</td>
<td>0.06942</td>
<td>40°</td>
<td>72</td>
<td>58</td>
</tr>
<tr>
<td>q7i80_low-e 7</td>
<td>7</td>
<td>(-0.787846, 0, 0.138919)</td>
<td>19.5</td>
<td>1.0387</td>
<td>0.2</td>
<td>0.07954</td>
<td>80°</td>
<td>60</td>
<td>46</td>
</tr>
<tr>
<td>q7i00_high-e 7</td>
<td>7</td>
<td>(0, 0, 0.8)</td>
<td>21.125</td>
<td>0.8617</td>
<td>0.3</td>
<td>0.20320</td>
<td>0°</td>
<td>60</td>
<td>45</td>
</tr>
<tr>
<td>q7i40_high-e 7</td>
<td>7</td>
<td>(-0.51423, 0, 0.612836)</td>
<td>21.125</td>
<td>0.8617</td>
<td>0.3</td>
<td>0.20690</td>
<td>40°</td>
<td>55</td>
<td>42</td>
</tr>
<tr>
<td>q7i80_high-e 7</td>
<td>7</td>
<td>(-0.787846, 0, 0.138919)</td>
<td>21.125</td>
<td>0.8617</td>
<td>0.3</td>
<td>0.21950</td>
<td>80°</td>
<td>44</td>
<td>32</td>
</tr>
</tbody>
</table>

spans the full range of orbits from the perspective of motion in Kerr, from equatorial to near-equatorial to nearly polar orbits. Figure 5.1 plots the trajectories for portions of two of our $q = 7$, high eccentricity orbits.

In Table 5.1 we present the full range of simulation parameters. The number of radial oscillations before merger $N_r$ is estimated by counting the number of maxima in the coordinate separation $r$ of the centers of the black holes, following the initial junk phase. The number of orbits $N_\phi$ is estimated by integrating the orbital frequency $\Omega(t)$ defined in (5.10) below over the entire inspiral. For our precessing runs, this $N_\phi$ does not correspond to the number of azimuthal cycles in the fixed simulation coordinates, since $\Omega(t)$ is the angular velocity in the instantaneous, precessing orbital plane. Computation of the simulation eccentricity $e$ [and thus its initial eccentricity $e_0 = e(t = 0)$] is described below in section 5.4.2; the expression for $e(t)$ is given by (5.14). The target eccentricity $e_N$ turns out to produce an overestimate of the initial eccentricity $e_0$, but one which is stable across our simulations.

### 5.4.2 Dynamics of simulated binaries

In order to connect our simulations with analytic theory we need to first construct quantities that can be used to extract fundamental frequencies. Numerical simulations yield the position vectors $x_1(t)$ and $x_2(t)$ of the Cartesian coordinate-centres of the holes, and a spin-vector $\chi_1(t)$ describing the spin angular momentum of the more massive hole.

From the position vectors we compute a radial separation vector $r(t) \equiv x_1(t) - x_2(t)$
Figure 5.2: Coordinate separations of our BBH simulations as a function of simulation time. The simulations are grouped into four panels according to mass-ratio (left, $q = 5$; right, $q = 7$) and eccentricity (top, low eccentricity; bottom, high eccentricity). Each panel shows runs at the three different inclinations $i$. 

Low ecc

<table>
<thead>
<tr>
<th>$r/M$</th>
<th>$t/(10^3M)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0^\circ$</td>
<td>0 5 10 15 20</td>
</tr>
<tr>
<td>$i = 10^\circ$</td>
<td>0 5 10 15 20</td>
</tr>
<tr>
<td>$i = 20^\circ$</td>
<td>0 5 10 15 20</td>
</tr>
</tbody>
</table>

High ecc

<table>
<thead>
<tr>
<th>$r/M$</th>
<th>$t/(10^3M)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0^\circ$</td>
<td>0 3 6 9 12 15 18</td>
</tr>
<tr>
<td>$i = 40^\circ$</td>
<td>0 3 6 9 12 15 18</td>
</tr>
<tr>
<td>$i = 80^\circ$</td>
<td>0 3 6 9 12 15 18</td>
</tr>
</tbody>
</table>
along with its magnitude $r \equiv |r|$. The separation $r$ has no invariant meaning, but it illustrates the dynamics of the inspiral; we thus plot it in Figure 5.2. The number of extrema of $r$ indicate the number of apastron and periastron passages, and the decline in eccentricity during the inspiral is evident in the decreasing amplitude of the oscillations of $r$.

For the equatorial runs we do not work with $r$ directly. Instead we compute from it an orbital frequency \cite{71}

$$\Omega(t) \equiv \frac{|\mathbf{r} \times \dot{\mathbf{r}}|}{r^2}, \quad (5.10)$$

from which we are able to extract the fundamental frequencies $\Omega^\phi(t)$ and $\Omega^r(t)$. In practice, we compute $\dot{\mathbf{r}} = d\mathbf{r}/dt$ with a 3rd order Savitzky-Golay filter windowed over groups of seven points \cite{267}. Our time coordinate is asymptotically inertial \cite{62}, justifying our identification of simulation coordinate frequencies with those measured by asymptotic, inertial observers.

In Figure 5.3 we plot $\Omega(t)$ for our simulations, which clearly shows modulations due
to the eccentricity. The modulation amplitude decreases throughout the inspiral as eccentricity is radiated away. The modulations are not simple sinusoidal oscillations about a chirping mean, but instead exhibit sharper peaks at periastron, an effect more pronounced at higher eccentricities.

Our analysis of eccentric motion is based on the oscillatory features of \( \Omega(t) \). For non-dissipative orbits, one would simply utilize the extrema of \( \Omega(t) \). However, as shown in Figure 5.3, GW-driven inspiral adds an overall monotonic trend to the radial motion of the binary, so that extrema of \( \Omega(t) \) do not correspond precisely to the reversals of the underlying oscillatory behavior. Should the growth rate of the monotonic trend ever exceed the range of the growth rate of the oscillatory one, as can easily happen at late inspiral, or even early inspiral when eccentricity is low, \( \Omega(t) \) will no longer display extrema at all.

We begin by computing the maxima and minima of \( \Omega(t)^3 \). We denote the times where \( \Omega(t) \) takes its maximum and minimum values as \( t_i^+ \) and \( t_i^- \), respectively, where \( i \) labels successive maxima/minima. For a conservative orbit these extrema are precisely the points of physical interest and the procedure is complete.

For inspiraling orbits, we apply a refinement procedure based on the subtraction of the envelope of the upper extrema. This procedure is motivated by analyzing a simple model in 5.8.2 and is illustrated in Figure 5.4. We compute a spline-fit to the extrema \( \{(t_i^+, \Omega(t_i^+))\} \), and subtract this fit from \( \Omega(t) \). This results in the first-stage (\( N=1 \)) “envelope subtracted” frequency \( \Omega_{ES}^{(1)}(t) \), cf. the lower panel of Figure 5.4. We now iterate this procedure: Find the abscissas of the maxima of \( \Omega_{ES}^{(1)}(t) \); compute a spline-fit to the respective points in \( \Omega(t) \); subtract to generate the \( N=2 \) envelope subtraction, \( \Omega_{ES}^{(2)}(t) \). We iterate the procedure \( N_f \)-times until no more change occurs to within machine precision (typically \( N_f = 8 \)). We redefine the extremal times \( t_i^+ \), \( t_i^- \) as the maxima/minima of \( \Omega_{ES}^{(N_f)}(t) \). We finally define \( \Omega^\pm(t) \) as the spline-interpolant to \( \{(t_i^\pm, \Omega(t_i^\pm))\} \). (In practice, we are most interested in \( \Omega^+ \) when computing radial frequencies, since the sharper peaking of \( \Omega(t) \) at periastron allows the former to be located more accurately.) Envelope-subtraction generates a spline-interpolant \( \Omega^+(t) \) which appears tangent to the original \( \Omega(t) \), whereas the original envelope passing through the bare peaks would cross it (cf. the upper right inset of Figure 5.4). During our analysis of inclined runs in section 5.6 we use an envelope-subtracted separation

\[
    r_{ES}(t) \equiv r(t) - r^+(t),
\]

Extrema are found in practice by a quartic spline fit to the discretely sampled \( \Omega \) near each extremum, and computing the extremum of the spline, which we denote by \( \Omega_i^+ \) and \( \Omega_i^- \).
Figure 5.4: Illustration of refinement procedure applied to $\Omega$ from (5.10) computed from our $q7_{i80}$ high-e run. The top panel shows $\Omega$ itself, while the bottom shows the same after $N = 1$ and $N = 8$ iterations of envelope subtraction, when the location of the maxima are stable up to machine precision. The refined curve is computed by subtracting a 4th order spline interpolation of the maxima (the “envelope”) from the original curve; each iteration denotes a subsequent envelope subtraction from the result of this procedure. This results in an envelope which, rather than passing through the maxima of the original curve, lies tangent to it, as seen in the inserts. The difference between the maxima of each iteration is due to dissipation and thus becomes more pronounced further into the inspiral. The refinement procedure also sometimes finds additional peaks which were saddle points in the original curve (note the differences between the envelopes as well as the $\Omega_{ES}^{(N)}$ curves close to the end of the inspiral).
Figure 5.5: Eccentricity vs. orbital frequency for our numerical simulations (solid lines). The simulations are grouped into four panels according to mass-ratio (left, \(q = 5\); right, \(q = 7\)) and initial eccentricity (top, low eccentricity; bottom, high eccentricity). Dashed lines show the eccentricity decay predicted by leading-order post-Newtonian formulae [239, 240], starting from the initial eccentricity \(e_0\) of each binary.

which is generated with the identical procedure, substituting \(r(t)\) for \(\Omega(t)\) above.

With the output quantities of envelope-subtraction, we can now define several more useful quantities. First, the average orbital frequency in the \(i\)-th radial oscillation period is given by

\[
\langle \Omega \rangle_i \equiv \frac{1}{t_{i+1}^+ - t_i^+} \int_{t_i^+}^{t_{i+1}^+} \Omega(t) dt,
\]

and we assign the values \(\langle \Omega \rangle_i\) to the midpoint times

\[
\tilde{t}_i^+ \equiv \frac{t_i^+ + t_{i+1}^+}{2}.
\]
Furthermore, we define eccentricity using the Keplerian formula

$$e(t) \equiv \frac{\sqrt{\Omega^+(t)} - \sqrt{\Omega^-(t)}}{\sqrt{\Omega^+(t)} + \sqrt{\Omega^-(t)}}. \quad (5.14)$$

We define $e(t)$ based on orbital frequency rather than the separation since the extrema of the former can be located with marginally better accuracy.

For the inclined runs, we also need analogs for the Boyer-Lindquist coordinate $\theta$. We take $\theta$ to be the angle between $\chi_1$ and $r$; thus

$$\cos \theta \equiv \frac{\chi_1 \cdot r}{\chi_1 r}. \quad (5.15)$$

We view the “equatorial plane” as that normal to $\chi_1$. The coordinate $\theta$, or alternatively $\cos \theta$, oscillates between maximum and minimum values during the orbit, and this motion is the basis of our definition of the polar frequency of the inclined binaries discussed in section 5.6.

### 5.4.3 Plots/Eccentricity decay

As a first result of our definitions in Section 5.4.2, we consider the decay of eccentricity for equatorial binaries. Figure 5.5 plots $e(t^+)$ at the midpoint times $t^+$ versus $\langle \Omega \rangle_i$. This Figure illustrates the fast decay of eccentricity in a GW-driven inspiral, as first computed by [239, 240]. Figure 5.5 also includes a direct comparison with the leading-order post-Newtonian (PN) results of [239, 240], through a parametric plot of $\langle \Omega \rangle$ as a function of eccentricity, cf. equation (5.45) derived in 5.8.3. This leading PN result is independent of the inclination of the binary, and gives the evolution of the eccentricity starting from its initially measured value $e_0$ at the initially measured orbital frequency $\langle \Omega \rangle_0$, which differs from simulation to simulation. The agreement for eccentricity decay is quite remarkable, generalizing results in [218] to the larger eccentricities considered here.

### 5.5 Equatorial binaries

In this section we extract the two fundamental frequencies $\Omega^r$ and $\Omega^\phi$ for our eccentric, equatorial simulations. For these binaries, the frequencies have simple definitions in terms of finite-time averages over radial passages and can be extracted cleanly once we account for dissipation.
5.5.1 Frequencies for equatorial orbits

Fourier-based methods of frequency extraction are inaccurate over the short timescales forced upon us by by gravitational dissipation (see 5.8.4). We therefore rely on time-averages or period measurements between extrema of the orbital frequency $\Omega(t)$, cf. (5.10).

For circular, equatorial, conservative orbits $\Omega(t)$ is constant and equals the frequency $\Omega^\phi$ of the motion in the orbital plane. Adding dissipative inspiral will cause $\Omega(t)$ to increase monotonically with time.

Including eccentricity but not dissipation yields a periodic $\Omega(t)$ whose time-average is the constant $\Omega^\phi$ of the analogous circular, conservative case. These periodic oscillations track the radial motion of the orbit, with successive maxima $\Omega^+_i$ at periastron and minima $\Omega^-_i$ at apastron. These oscillations will not be symmetric about their mean, instead peaking near periastron. This foils attempts at frequency extraction using rolling fits [188, 189, 218] which have been used in the past to extract orbital frequencies in low-eccentricity BBH simulations. It is difficult to find suitable fitting-functions at higher eccentricities without introducing unacceptable model-dependence to the procedure.

Based on the envelope-subtracted maxima of the orbital frequency $\{(t^+_i, \Omega^+_i)\}$ (cf. section 5.4.2), we define the radial frequency $\Omega^r$ through the period between successive maxima,

$$\Omega^r_i \equiv \frac{2\pi}{t^+_{i+1} - t^+_i}. \quad (5.16)$$

Further, we define the azimuthal frequency as the average orbital frequency (5.12),

$$\Omega^\phi_i \equiv \langle \Omega \rangle_i, \quad (5.17)$$

which holds exactly for equatorial Kerr orbits. For each cycle we assign the values of $\Omega^r_i$ and $\Omega^\phi_i$ to the orbital midpoint times $\tilde{t}^+_i$, equation (5.13). The $r$-$\phi$ precession rate is then

$$K_{r\phi}^i \equiv \frac{\Omega^r_i}{\Omega^\phi_i}, \quad (5.18)$$

which we parametrize as a function of $\Omega^\phi_i$. Our choice to define frequencies over one radial oscillation period is ideal for equatorial inspirals, as it minimizes biasing by dissipation.

In order to compare the precession rate $K_{r\phi}$ to the precession rate of an eccentric orbit in Kerr, we must identify both a particular Kerr spacetime and a particular geodesic orbit to compare to. In the self-force formalism, the spacetime is expanded around a Kerr solution with mass $M_{Kerr}$ equal to that of the larger black hole, and not the total mass of the system. As such, we set the Kerr mass equal to the mass of the larger black hole.
$M_{\text{Kerr}} = m_1$ and the spin parameter to $a_{\text{Kerr}}/M_{\text{Kerr}} = \chi_1$. With the Kerr spacetime fixed, two further parameters are needed to identify a particular equatorial geodesic. Since we wish to compare $K^{r\phi}$ to an equivalent test orbit, we cannot use both $\Omega^r$ and $\Omega^\phi$ to select our reference orbit. We choose to identify our test orbit using $\Omega^\phi$ and the eccentricity $e(\Omega^\phi)$ computed using (5.14).

To compute the geodesic precession rate, we numerically invert the procedure for finding $\Omega^\phi(p,e,i = 0)$ from [270] to find $p$ as a function of $\Omega^\phi$ when restricted to the eccentricities $e(\Omega^\phi)$. This allows us to obtain analytic predictions $K^{r\phi}_{\text{Kerr}}(m_1\Omega^\phi_i,e_i)$ for the midpoint times $\hat{t}_i^+$ at which we extract $K^{r\phi}$ from our simulations. We subtract this analytic prediction from our extracted precession rates at each $\Omega^\phi_i$. We can then reparameterize these differences in terms of any other quantity defined at the same $\hat{t}_i^+$.

This procedure is not ideal: the eccentricity estimator (5.14) is chosen to give stable results and does not correspond to the Boyer-Lindquist coordinate eccentricity used in Kerr. However, the frequencies are only weakly dependent on $e$ at small eccentricity (e.g. [270]) and so this approximation impacts $K^{r\phi}$ also weakly. Ideally, we would compute a third scalar quantity $f$, averaged over the orbit, parametrize it by our measured $\Omega^r$ rather than the eccentricity, and compare the extracted time series $f_i$ to analytic predictions for $f(\Omega^\phi,\Omega^r)$. An example would be the third, polar frequency $\Omega^\theta$. While $\Omega^\theta$ remains well-defined for equatorial orbits, we cannot in practice extract it from simulation without measurable polar motion. With no such third quantity available, we compare our eccentricity-dependent prediction to simulation, in order to get a qualitative understanding of the observed precession rate $K^{r\phi}$.

### 5.5.2 Results

We plot our extracted precession rates $K_i^{r\phi}$ along with the geodesic predictions as functions of azimuthal frequency $M\Omega^\phi_i$ in Figure 5.6. Several numerical resolutions are plotted, and we find agreement in our precession rates across resolutions. In addition, it is apparent that the finite mass corrections increase the magnitude of general relativistic precession, by decreasing $\Omega^r$ as compared to $\Omega^\phi$. Higher eccentricities also enhance the precession rate. Figure 5.6 enables identification of $r$-$\phi$ resonances, which are marked with horizontal lines. Our binaries plunge before the high-order $r$-$\phi$ resonances which are expected to generate resonant kicks. Thus, although the analytical results of [309]

---

4 An example of a third invariant quantity would be the orbit-averaged redshift [20,47], which can be computed from numerical simulations with the techniques developed in [332]. The simulations presented here were begun before [332], and therefore the necessary data for computing the redshift was not preserved.
Figure 5.6: Precession rates $K^{r\phi}$ as a function of $M\Omega^\phi$ extracted from our equatorial simulations (thick solid, dashed, and dotted lines) and as predicted for a Kerr black hole of mass $M_{\text{Kerr}} = m_1$ and the same eccentricity evolution (thin lines with symbols). We plot multiple resolutions for our simulations as indicated. Left: Precession of our low-eccentricity simulations. Right: Precession of our high-eccentricity simulations.

indicate these kinds of resonant kicks could be promising at mass ratios comparable to ours, we see that in reality the rate of inspiral is too high and significantly higher mass ratios are required to access the relevant resonances.

In the left panel of Figure 5.7 we plot the relative difference between the numerical precession rates and geodesic predictions, $\Delta K^{r\phi}_i = K^{r\phi}_{\text{Numeric},i} - K^{r\phi}_{\text{Kerr},i}$, normalized by the numerical $K^{r\phi}_i$. We see that the geodesic prediction captures the precession rate well for our binaries, up to corrections of less than ten percent. As expected, the $q = 7$ simulations are closer to the geodesic predictions than the $q = 5$ simulations. Inspecting each mass ratio separately, we see that doubling the eccentricity has less than a percent impact on the relative differences.

Finally, we use our simulations to extract the $O(1/q)$ corrections to the geodesic precession rate, i.e. the leading conservative SF correction to $K^{r\phi}$. In the right panel of Figure 5.7, we rescale the relative differences by $q$, and plot our results against $m_1\Omega^\phi_i$. This guarantees that we are expanding our results around the same test mass limit in all cases. Remarkably, we see that after this rescaling the $q = 5$ and $q = 7$ curves are nearly identical in both the high-eccentricity and low-eccentricity cases. This implies several things. First, that the higher order SF effects must be quite small, since a priori an $O(1/q^2)$ correction would generate a similar split in the rescaled $q = 5$ and $q = 7$ curves as seen in the left panel of Figure 5.7. Next, it must be true that the spin-dependence of the $O(1/q)$ SF correction is almost entirely captured by the prefactor, which is normalized out in each case. Finally, comparing the high- and low-eccentricity cases to each other, we see the same limited dependence on eccentricity as discussed for the left hand panel.
5.6 Inclined binaries

In this section, we extract the frequencies of motion for our eccentric, inclined simulations. In these cases, precession of the orbital plane and the black hole spin introduces a non-trivial dependence between all three of the characteristic frequencies. This complicates our procedure, but we are able to cleanly extract $K^{r\theta}$ and compare to Kerr geodesics. We find that our binaries pass through low order resonances in $r$-$\theta$ motion, and we investigate the fluxes of energy and angular momentum from our simulations to search for the imprint of these resonances.

5.6.1 Definition of frequencies

For inclined runs, there are two periodic modulations which both imprint themselves on the dynamics, cf. (5.3) and (5.4). When choosing averaging-periods, we cannot honour both periodicities simultaneously. Empirically, we find that windows over the radial rather than polar periods yield smoother frequencies. For instance, $\Omega$ varies much more

Figure 5.7: Comparison between the precession rates for our equatorial binary inspirals and Kerr geodesic theory. Specifically, we plot the differences $\Delta K^{r\phi}$ between the numerical extraction and geodesic prediction for the $r$-$\phi$ frequency ratio, divided by the numerical $K^{r\phi}$.

All together, we conclude that the periastron precession rate takes the form

$$K^{r\phi}(m_1 \Omega^\phi) = K^{r\phi}_{\text{Kerr}}(m_1 \Omega^\phi) \left[1 + \frac{1}{q} \Delta K^{r\phi}_{\text{SF}} + O\left(\frac{1}{q^2}\right)\right].$$

where $\Delta K^{r\phi}_{\text{SF}}$ is nearly independent of spin and eccentricity at modestly large spins and low eccentricities, and the further $O(1/q^2)$ terms are numerically small.
Figure 5.8: Plot of $\cos \theta$ versus $t$ (left) and $r$ (right) for a Kerr geodesic with semilatus rectum $p = 20M$, eccentricity $e = 0.9$, $i = 36^\circ$, and spin parameter $\chi = 0.9$ (top), compared with our high mass ratio and eccentricity $i = 80^\circ$ BBH simulation (bottom). In the BBH case we use the envelope subtracted separation $r_{ES}$, computed as described in section 5.4. Note in particular the strong dependence of $\partial_t \cos \theta$ upon $r$, most obviously visible in the Kerr phase plot. This strongly modulates the polar motion in time, resulting in the beating effect visible in both time series plots. This effect necessitates a more complex frequency-extraction strategy than taking simple peak-to-peak periods of the polar motion.

strongly between periastron and apastron than for different values of $\theta$. These findings depend on the eccentricities considered here: we expect that if eccentricity were reduced at fixed inclination, eventually the polar oscillations would become more important than the radial oscillations.

The radial motion couples to the polar motion through spin-orbit coupling via terms proportional to $\hat{\chi}_1 \cdot \hat{r} = \cos \theta$, which results in a $\cos \theta$-dependent modulation to the envelopes of $\Omega(t)$ and $r(t)$. This introduces some oscillation to both $\Omega'$ defined in (5.16) and especially to the eccentricity $e$ (5.14). Nevertheless these effects are relatively small compared to those which would be introduced by choice of a different window for the $\Omega'$ computation, and we continue to use the same relation as in the equatorial case: $\Omega'$ is the reciprocal of the periods between maxima of the orbital frequency $\Omega(t)$, (5.16).
We now turn to the polar frequency $\Omega^\theta$, which we would like to define in terms of the binary black hole system’s polar motion $\theta(t)$. This angle $\theta(t)$ shows clear modulations due to the interdependence of the radial and polar motion: roughly, $\theta(t)$ varies most quickly near periastron. While the resultant modulations to $\theta(t)$ are most pronounced for highly eccentric orbits such as the Kerr geodesic shown in the upper panel of Figure 5.8, they are still visible by eye at the eccentricities accessible to numerical simulations (cf. the lower panel of Figure 5.8). A straightforward definition of polar frequency based on extrema of $\theta(t)$, while satisfactory for conservative orbits once averaged over infinite polar cycles, suffers from substantial interval-to-interval variations during a simulation, depending on the radial phase at successive $\theta(t)$ extrema.

The most obvious way to account for the dominant dependence on separation is to measure the polar frequency over intervals that begin and end at fixed radial phase. While we do not have access to such a phase in general, any reasonable definition of one will attain a fixed value at periastron. We therefore employ the same periastron-to-periastron intervals as those we used to define $\Omega^r$ (5.16). As an additional advantage, this choice results in an $\Omega^\theta$ defined at the same midpoint times $\tilde{t}_i^+$ as the other frequencies, such that comparisons do not require an interpolation.

Because the radial and polar frequencies are distinct, the periastron-to-periastron intervals $[t_i^+, t_{i+1}^+]$ do not correspond to an integer number of polar oscillations. To account for this we define a polar phase $\chi^\theta(t)$ through equation (5.6), and we define $\cos \theta_{\text{min}}$ using the maximal envelope of $\cos \theta(t)$, interpolated to the time of interest.

The function $\chi^\theta(t)$ is made monotonic and continuous by suitable choice of quadrant when inverting $\cos \chi^\theta$ followed by suitable additions of multiples of $2\pi$. We then define

$$\Omega_i^\theta \equiv \frac{\chi^\theta(t_{i+1}^+) - \chi^\theta(t_i^+)}{t_{i+1}^+ - t_i^+}. \tag{5.20}$$

In 5.8.4 we show that this definition applied to Kerr limits to the exact polar frequency, with sub-percent error over single-cycle windows.

Finally, the azimuthal frequency $\Omega^\phi$ presents the most difficulties. For inclined Kerr orbits, the averaged orbital frequency $\langle \Omega \rangle$ differs from the azimuthal frequency $\Omega^\phi$. One solution would be to define $\Omega^\phi$ in terms of some azimuthal angle $\phi$, in a similar manner as in (5.20). Such a definition is possible, but all our attempts have resulted in an $\Omega^\phi$ which either fails to converge properly for Kerr orbits or which oscillates unacceptably wildly for single-cycle windows. To appreciate the difficulty, note for example that the “orbital plane” of the BBH simulation might reasonably be defined as the plane orthogonal either to the primary spin vector $\chi_1$ or to the total angular momentum vector $J$. Both choices
recover the desired limit for \( q \rightarrow \infty \), but are different planes at finite \( q \).

On the other hand \( \langle \Omega \rangle \) does furnish a fairly smooth frequency which is itself of some dynamical interest. We therefore report \( \langle \Omega \rangle \) computed via (5.12) for the inclined orbits as well.

### 5.6.2 Precession rates

From the frequencies \( \Omega^r, \Omega^\theta \) and \( \langle \Omega \rangle \) we compute precession rates as frequency ratios \( K^{ab} = \Omega^a / \Omega^b \). The results are shown in Figure 5.9. As in section 5.5, we plot against a frequency, \( M\Omega^\theta \), in order to mitigate any gauge effects. Since we cannot cleanly extract \( \Omega^\phi \), we plot the ratios \( K^{r(\phi)} \equiv \Omega^r / \langle \Omega \rangle \) and \( K^{\theta(\phi)} \equiv \Omega^\theta / \langle \Omega \rangle \) in addition to \( K^{r\theta} \). We do not have analytic predictions for the ratios involving \( \langle \Omega \rangle \), but we can see some general trends. As the binary sweeps to higher frequencies, periastron precession becomes stronger as \( \Omega^r \) lags further behind the other frequencies. Comparing the higher eccentricity to the lower eccentricity simulations at fixed \( \Omega^\theta \), we see that \( K^{ab} \) is smaller in the higher eccentricity cases. This is most apparent early in the inspirals (smaller \( \Omega^\theta \)), when there is a larger...
difference in $e$ between the cases; for all the $K^{ab}$, the solid lines lie at smaller values than the dashed, although the differences are small. This is true also for the Kerr predictions (bottom right of Figure 5.9).

There is no clear effect of inclination on the precession rates $K^{r\theta}$ and $K^{r\langle\phi\rangle}$, except for $i = 80^\circ$, where they are systematically smaller, i.e. periastron advance is more pronounced. Meanwhile, the effect of increasing inclination on $K^{\theta\langle\phi\rangle}$ is clearly visible: at higher inclinations, $\langle\Omega\rangle$ becomes more and more nearly equal to $\Omega^\theta$, and so their ratio approaches unity as we approach polar orbits.

Figure 5.9 shows large oscillations in the extracted precession rates $K^{r\theta}$ and $K^{r\langle\phi\rangle}$ for the simulations $q7_{i40}$low-e and $q7_{i80}$low-e. These oscillations can be traced to distinct features in the separation $r(t)$ for these two simulations. Compared to the other simulations considered here, $r(t)$ shows extra modulations for these two cases. We discuss this in more detail in 5.8.4. We presently do not understand the origin of these additional features, nor the reason why they only appear in the simulations $q7_{i40}$low-e and $q7_{i80}$low-e. All $q = 7$ simulations were run with identical source-code revision and configuration files.

Figure 5.10 illustrates the relative differences between the geodesic predictions and our numerical precession rates. We focus on $K^{r\theta}$, which we can compare to analytic theory. The top left panel of Figure 5.10 shows the $q = 5$ data, where we find no discernible difference between the inclinations $i = 10^\circ$ and $i = 20^\circ$. Further, as in the case of the equatorial inspirals, the difference between low eccentricity and high eccentricity is mild. The top right panel shows the high eccentricity case for the $q = 7$ inspirals, and again we see only a mild dependence on the inclination, although it is discernible. The low $e$ runs for $q = 7$, unfortunately, are dominated by the large modulations mentioned above and do not add any further insights.

The lower panels of Figure 5.10 plot the rescaled residual differences, with the lower eccentricity inspirals on the left and the higher eccentricities on the right. As seen before in Figure 5.7, the fact that all the lines are nearly on top of each other indicates that the $O(1/q^2)$ corrections are unexpectedly small, cf. equation (5.19). Even in the case of the $q = 7$, low eccentricity runs, we see that the midline of the large modulations agrees well with the $q = 5$ scaled residuals. Thus, the curves here roughly give the leading SF conservative corrections to the $r-\theta$ precession rate. Further, we see that the inclination, spin, and eccentricity dependence of this correction is mostly captured by scaling out the geodesic results.
Figure 5.10: *Top:* Comparison of the numerical extraction of $K^{r\theta}(M\Omega_\theta)$ to geodesic theory, for each of $q = 5$ and $q = 7$ (higher eccentricity simulations only). The frequencies for the low eccentricity $q = 7$ simulations display large modulations, as can be seen in Fig. 5.9. *Bottom:* Plot of $\Delta K^{r\theta}/K^{r\theta}$ rescaled by $q$ and plotted against $m_1\Omega_\theta$ for each simulation. The left panel features the lower eccentricity simulations, and the right panel features the higher eccentricity simulations.
5.6.3 Resonances

The frequencies computed in section 5.6 can be immediately exploited to detect resonances. We focus on the lowest order $r$-$\theta$ resonances our simulations achieve, which are the 5:6, 4:5, 3:4 and 2:3 resonances. In particular, self-force calculations using PN models [53,118] have highlighted the importance of the 2:3 resonance.

The rough locations of the coordinate resonances can be identified from Figure 5.9, where the relevant values of $K^{ab}$ are highlighted with horizontal lines. A somewhat better estimate can be obtained by solving the equation $k\Omega^\theta(t) - n\Omega^r(t) = 0$ for integers $n, k$, where $n/k \leq 1$ gives the value of $K^{ab}$ at resonance. This avoids the noise in the quotient. Solving this equation gives us an estimate for the simulation time $t_{\text{res}}$ at which a given resonance is encountered. Interpolating $\Omega^\theta$ onto this value gives an estimate of the corresponding polar frequency at resonance, $\Omega^\theta_{\text{res}}$.

Following [260] we estimate the time spent on $r$-$\theta$ resonance using the following procedure. First, we construct a resonant phase $\Phi_{kn}(t)$

$$\Phi_{kn}(t) = \int_{t_0}^{t} \left[ k\Omega^\theta(t') - n\Omega^r(t') \right] dt', \quad (5.21)$$

where the arbitrary reference time $t_0$ is chosen after initial transients of the numerical simulation\(^5\). The significance of the phases $\Phi_{kn}$ arises from black hole perturbation theory, where the effect of the dissipative self-force upon the time derivative of some Kerr constant of motion $C$ expands into Fourier modes $k$ and $n$ of the form

$$\frac{dC}{dt} = \dot{C}_{00} + \sum_{k=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} \dot{C}_{kn} e^{-i\Phi_{kn}(t)}, \quad n, k \neq 0. \quad (5.22)$$

The exponential terms in most cases average to zero since they oscillate rapidly compared to the overall evolution of $C$. At resonance, however, the phase is constant and the respective term contributes secularly to $dC/dt$.

A representative resonant phase for each order $k : n$ is shown in Figure 5.11. Still following [260], we say the system is near resonance when $\Phi_{kn}$ differs from its resonant value by no more than 1 radian. This dephasing is indicated in Figure 5.11 by the dashed vertical lines; our simulations satisfy this condition for durations of a few $1000M$. The duration $\Delta t_{\text{res}}$ and frequency range $\Delta \Omega^\theta_{\text{res}}$ during which our simulations stay on resonance can be readily determined from $\Phi_{kn}$. We estimate the number of radial and polar cycles

\(^5\)In practice, we evaluate $k\Omega^\theta_i - n\Omega^r_i$, spline-interpolate this discrete time series, and integrate the interpolant.
Figure 5.11: Pictorial analysis of $r$-$\theta$ resonances for four select resonances. For each example, a block of three graphs is presented, titled by the simulation and the resonance under consideration. Each block is organized as follows: Top graph: Resonant phase $\Phi_{kn}$ with vertical solid lines marking the resonance, vertical dashed lines bounding one radian in resonant phase in either direction, and black dots indicating midpoint times, $\tilde{t}_i^+$, cf. (5.13). The resonant phase $\Phi_{56}(t)$ in the top-left panel is plotted at three different numerical resolutions. Bottom left graph: Orbital trajectory during the resonance using two line-styles, solid lines represent one resonant cycle centered on the resonance (marked by a circle); dashed lines represent the remaining evolution within the resonant window. Bottom right graph: Trajectory with inspiral-motion removed, by using the envelope-subtracted radius $r_{ES}$ (5.11).
Figure 5.12: Gravitational wave fluxes during resonances. Each row of plots corresponds to one numerical simulation as indicated in the title of each panel. The columns correspond to three gravitational wave fluxes. Left: Angular momentum flux orthogonal to primary spin direction $\hat{\chi}(t)$. Middle: Angular momentum flux tangential to $\hat{\chi}(t)$. Right: Energy flux. Resonances are indicated by the shaded areas. The fluxes are shown at three resolutions, with differing linestyles; on the scale of the plot, however, these are indistinguishable in the plots except close to merger.

spent on resonance by counting the number of relevant coordinate peaks within the resonance. These data are collected in Table 5.2. Inspection of Table 5.2 reveals that, with the 2:3 resonance excepted, our simulations remain on resonance long enough to trace out one or even two full resonant cycles. (cf. the coordinate plots in Figure 5.11). On resonance, the $r$-$\theta$ motion of a geodesic is a Lissajous Figure; Figure 5.11 includes the corresponding plots for the full BBH simulation, highlighting that the character of the Lissajous Figures are preserved.

As Table 5.2 and Figure 5.11 indicate, our inspirals remain on resonance for one to two resonant cycles. This may lend some hope that secular accumulations from terms like those in (5.22) might be directly visible, leading to a noticeable change in the evolution of constants of motion ($\mathcal{E}, \mathcal{L}_z$, and $\mathcal{Q}$) during the inspiral. Returning to equation (5.22),
Table 5.2: The \( r-\theta \) resonances identified in our simulations. Given is the order of the resonance, the polar frequency \( \Omega^\theta \) at resonance, the duration in terms of time \( \Delta t_{\text{res}} \) and polar frequency \( \Delta \Omega^\theta_{\text{res}} \), and the number of radial cycles \( N_r \) and polar cycles \( N_\theta \) during the resonance. For many of the 2:3 and some of the 6:7 resonances, the resonances were detected, but occurred too near plunge for their width to be measured; in those cases we report the frequency at resonance and mark the other columns n/a.

<table>
<thead>
<tr>
<th>Run</th>
<th>Order</th>
<th>( M\Omega^\theta_{\text{res}} )</th>
<th>( \Delta t_{\text{res}} )</th>
<th>( M\Delta \Omega^\theta_{\text{res}} )</th>
<th>( N_r )</th>
<th>( N_\theta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>q5_l0_l0</td>
<td>6:7</td>
<td>0.0264</td>
<td>4540</td>
<td>0.00787</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>5:6</td>
<td>0.0341</td>
<td>2921</td>
<td>0.00506</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4:5</td>
<td>0.0385</td>
<td>1737</td>
<td>0.00301</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3:4</td>
<td>0.0408</td>
<td>920.3</td>
<td>0.00159</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>q5_l20_l0</td>
<td>6:7</td>
<td>0.0268</td>
<td>4646</td>
<td>0.00876</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>5:6</td>
<td>0.0353</td>
<td>3026</td>
<td>0.00571</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4:5</td>
<td>0.0403</td>
<td>1792</td>
<td>0.00338</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3:4</td>
<td>0.0430</td>
<td>989.0</td>
<td>0.00187</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>q5_l10_h0</td>
<td>6:7</td>
<td>0.0209</td>
<td>4713</td>
<td>0.0144</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>5:6</td>
<td>0.0346</td>
<td>2982</td>
<td>0.00911</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4:5</td>
<td>0.0425</td>
<td>1757</td>
<td>0.00537</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3:4</td>
<td>0.0466</td>
<td>935.4</td>
<td>0.00286</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2:3</td>
<td>0.0531</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>q5_l20_h0</td>
<td>6:7</td>
<td>0.0184</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>5:6</td>
<td>0.0301</td>
<td>3088</td>
<td>0.00814</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4:5</td>
<td>0.0379</td>
<td>1821</td>
<td>0.00480</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3:4</td>
<td>0.0416</td>
<td>964.9</td>
<td>0.00254</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>q7_l0_l0</td>
<td>6:7</td>
<td>0.0299</td>
<td>5048</td>
<td>0.00780</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>5:6</td>
<td>0.0385</td>
<td>3253</td>
<td>0.00502</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>4:5</td>
<td>0.0437</td>
<td>2076</td>
<td>0.00320</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>3:4</td>
<td>0.0469</td>
<td>279.2</td>
<td>0.000431</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2:3</td>
<td>0.0515</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>q7_l0_l0</td>
<td>5:6</td>
<td>0.0230</td>
<td>5569</td>
<td>0.00880</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>4:5</td>
<td>0.0341</td>
<td>3817</td>
<td>0.00603</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>3:4</td>
<td>0.0393</td>
<td>2182</td>
<td>0.00345</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>2:3</td>
<td>0.0447</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>q7_l0_l0</td>
<td>6:7</td>
<td>0.0224</td>
<td>5317</td>
<td>0.0124</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>5:6</td>
<td>0.0360</td>
<td>3371</td>
<td>0.00784</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>4:5</td>
<td>0.0435</td>
<td>1958</td>
<td>0.00456</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>3:4</td>
<td>0.0475</td>
<td>954.9</td>
<td>0.00222</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2:3</td>
<td>0.0537</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>q7_l0_l0</td>
<td>4:5</td>
<td>0.0311</td>
<td>3506</td>
<td>0.00975</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>3:4</td>
<td>0.0421</td>
<td>1858</td>
<td>0.00517</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2:3</td>
<td>0.0469</td>
<td>815.3</td>
<td>0.00227</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
we see that the impact of passage through a resonance is that a subset of the oscillatory contributions to \( dC/dt \) momentarily “freeze out”, with \( \dot{\Phi}_{kn} \) passing through zero at the resonance for pairs \( (k, n) \) commensurate with the resonance. Near the resonance, the phases of these contributions can be approximated by Taylor expanding \( \Phi_{kn} \) around the time \( t_{\text{res}} \) when resonance is achieved (see e.g. [260])

\[
\Phi_{kn} \approx \Phi_{kn}(t_{\text{res}}) + \frac{1}{2}(k\dot{\Omega}^\theta - n\dot{\Omega}^r)|_{t_{\text{res}}}(t - t_{\text{res}})^2.
\]  

(5.23)

While the amplitude and sign of these resonant contributions to \( dC/dt \) depends on the coefficients \( \dot{C}_{kn} \) and the phase on resonance \( \Phi_{kn}(t_{\text{res}}) \), these terms have a distinct time evolution: their contribution is approximately symmetric about \( t_{\text{res}} \), and they contribute over a time scale of approximately the resonant width \( \Delta t_{\text{res}} \). This characteristic time evolution is clearly seen in the plots of \( \cos \Phi_{kn} \) in Figure 5.11 around each resonant passage. Thus the simplest way to diagnose whether a resonant passage impacts the dynamics is to search for modulations of \( dC/dt \) with this behavior. Meanwhile, the time integral of these terms results in a jump in \( C \) which accumulates over a period approximately equal to \( \Delta t_{\text{res}} \). The detailed time evolution of these contributions involves Fresnel integrals (e.g. [53]), and is antisymmetric across the resonance.

To investigate the possibility of resonances affecting the orbital evolution, we compute the instantaneous gravitational wave energy flux \( \dot{E}(t_r) \) and the gravitational wave angular momentum fluxes \( \dot{J}(t_r) \) as suitable products of spherical harmonic modes of the gravitational waveform [63] for all modes \( l \leq 8 \). These fluxes are given in terms of the retarded time

\[
t_r = t - R - 2M \ln \left( \frac{R}{2M} - 1 \right),
\]

(5.24)

where \( R \) is the gravitational wave extraction radius of each simulation (\( \sim 500M \)). For the angular momentum flux, we project parallel and orthogonal to the primary BH-spin direction \( \hat{\chi}(t) \),

\[
\dot{J}_{\parallel}(t_r) = \dot{J}(t_r) \cdot \hat{\chi}(t),
\]

(5.25)

\[
\dot{J}_{\perp}(t_r) = \left| \dot{\mathbf{J}}(t_r) - J_{\parallel}(t_r)\hat{\chi}(t) \right|.
\]

(5.26)

We use \( \dot{J}_{\parallel} \) as a proxy for the azimuthal angular momentum flux of the test body (corresponding to the constant of motion \( \mu \mathcal{L}_z \)), and \( \dot{J}_{\perp} \) as a very rough proxy for the evolution of the square root of the Carter constant \( \mu \sqrt{\mathcal{Q}} \).

In Figure 5.12 we plot the energy- and angular momentum-fluxes for three simulations with the time-intervals on resonance indicated. Figure 5.12 shows short-period modulations due to eccentricity, with sharp spikes near each periastron passage. However, no
additional modulations over the resonant timescales are discernible. We have also examined the angular momentum $J$ and total energy $E$ by integrating the fluxes, and see no excess build-up or deficit over the resonant widths.

The amplitude of any changes that occur during resonance depends on the resonant phase and it is possible that all of our simulations have phases that make resonant effects undetectable. Unfortunately, the resonant phase cannot be easily changed in a numerical simulation while keeping all other parameters constant. Since it is unlikely that all resonances are encountered with unfavorable phases, we interpret the absence of detectable resonant modulations as evidence that the mass-ratios and eccentricities under consideration are not extreme enough to exhibit strong resonant effects.

In 5.8.5 we estimate quantitatively the strength of resonant effects by fitting a sufficiently-stiff function to peak-to-peak averages over the gravitational wave fluxes averaged roughly over the resonant timescale. Since the chosen function is too stiff to capture resonant-timescale effects, resonant accumulations should be visible as jumps in the residuals over that timescale. Using this method on the most extreme cases of the $q_{7\times40}$ high-e and $q_{7\times80}$ high-e runs we bound resonant effects upon these runs from above at less than the 0.4% level.

### 5.7 Conclusions

We have presented a suite of binary black hole simulations performed with the SpEC code that aim to explore the relation between BBH systems and generic Kerr geodesics. Our simulations are eccentric and explore both precessing and nonprecessing configurations. The simulations cover 30–60 radial passages and multiple precession cycles, and are performed at comparatively high mass-ratios of $q = 5$ and $q = 7$.

As a first step to understanding these fully generic binary inspirals, we developed methods for extracting the instantaneous, fundamental frequencies of motion while partially accounting for dissipative effects. In the case of equatorial inspirals, these are the frequencies of radial motion $\Omega^r$ and azimuthal motion $\Omega^\phi$ averaged over a radial orbit. Here the frequencies can be extracted most cleanly, since the interdependence of the polar and the radial motion does not come into play and does not produce frequency modulations. The ratio of these frequencies provides the rate of periastron advance and is a physically measurable quantity. We have compared the precession rate of our eccentric binaries to geodesic orbits in Kerr, and extracted the leading self-force correction to this rate. Our results show that second-order SF corrections to these rates is small, and that much of the dependence on eccentricity and black hole spin can be absorbed into the
leading Kerr behavior.

In the case of our generic, precessing binaries, we can cleanly extract the fundamental frequencies of radial and polar motion, as well as the average orbital frequency $\langle \Omega \rangle$. The fundamental azimuthal frequency $\Omega^\phi$ differs from $\langle \Omega \rangle$ because motion in the orbital plane is a mixture of azimuthal and polar motions for precessing binaries. Unfortunately, it turns out that $\Omega^\phi$ is affected by interactions of the polar and radial motions to a large degree, cf. 5.8.4. Nevertheless, we have extracted the ratios of radial, polar, and average orbital frequencies from our simulations, cf. Figure 5.9. The ratio $K^\| = \Omega^r / \Omega^\theta$ can be predicted for Kerr geodesics, and in this case we have compared to analytic theory and extracted the SF corrections to our precession rates.

Finally, with our fully generic orbits we were able to identify low-order orbital resonances in the radial-polar motion, and show that our systems pass through resonances with $\Omega^r / \Omega^\theta = 5:6, 4:5, 3:4$ and $2:3$. Our simulations remain on resonance usually for more than one resonant cycle, and sometimes up to two resonant cycles. The $r-\theta$ resonances have been shown to have an impact on the adiabatic inspirals of extreme mass ratio systems [117, 160], and our study is the first to identify them in a numerical spacetime. However, at the mass ratios we have achieved we find no clear impact on the fluxes of energy and angular momentum of our systems (cf. Figure 5.12).

In two of our low-eccentricity simulations, strong residual oscillations in the extracted frequencies dominate the comparison. These modulations arise from small but visible variations in the radial separation of the binary, persist at several resolutions, and will be the subject of future investigation.

In this study we have focused on the dynamics of the black holes themselves. In future studies we will examine the gravitational waveforms produced by these systems, and explore the extraction of the fundamental frequencies directly from the waveform. Waveforms from our equatorial and fully generic system will serve to test how detectable these binaries are using current quasi-circular and eccentric waveform models, as well as help develop those models further. Waveforms from these binaries can also elucidate what constraints can be placed on eccentricities in future gravitational wave detections.

Future work will include extending our suite of simulations to higher mass ratios, and to cover a larger range of eccentricities and binary configurations. We will extract the redshift factor [332] from generic black hole binaries, which is analogous to the Lorentz factor of the black holes. With three frequencies and the Lorentz factor, we can imagine forming an explicit map between high mass ratio binaries and perturbed orbits in Kerr. Comparisons to post-Newtonian predictions for the frequencies of motion would supplement our SF-inspired comparisons to Kerr geodesics. In addition, SF predictions
in for Kerr orbits are developing rapidly (e.g. [310,311]), which will allow for a quantitative comparison to analytic approximations, as well as an extraction of higher-order SF effects.

In previous studies of circular orbits, it was suggested that re-expanding geodesic predictions about the symmetric mass ratio \( \nu = m_1 m_2 / M^2 \) and as functions of \( M \Omega^\phi \) would provide for faster convergence to the numerical results [188, 189]. With a SF prediction for the precession rates in our equatorial and generic binaries, we will be able test this promising idea in a new regime.

### 5.8 Appendices

#### 5.8.1 The Kerr metric

The Kerr metric expressed in Boyer-Lindquist coordinates \( x^\mu = (t, r, \theta, \phi) \) is given by

\[
ds^2 = -\left(1 - \frac{2Mr}{\rho^2}\right)dt^2 - \frac{4Mar\sin^2 \theta}{\rho^2}dtd\phi + \frac{\rho^2}{\Delta}dr^2 + \rho^2d\theta^2
+ \frac{\sin^2 \theta}{\rho^2} \left[(r^2 + a^2)^2 - a^2 \Delta \sin^2 \theta\right]d\phi^2.
\] (5.27)

where \( \rho^2 = r^2 + a^2 \cos \theta \), \( \Delta = r^2 - 2Mr + a^2 \), and \( M = M_{\text{Kerr}} \) in this appendix only.

The equations of motion for a test particle are given by (5.2)–(5.4), and the explicit forms of the potentials are (see e.g. [124])

\[
R(r) = \left[\mathcal{E}(r^2 + a^2) - a\mathcal{L}_z\right]^2 - \Delta \left[\left(a\mathcal{E} - \mathcal{L}_z\right)^2 + r^2 + \mathcal{Q}\right],
\] (5.28)

\[
\Theta(\theta) = \mathcal{Q} - \cos^2 \theta \left[a^2(1 - \mathcal{E}^2) + \mathcal{L}_z^2 \csc^2 \theta\right],
\] (5.29)

and

\[
T_r(r) = \frac{r^2 + a^2}{\Delta} \left[\mathcal{E}(r^2 + a^2) - a\mathcal{L}_z\right], \quad T_\theta(\theta) = -a^2\mathcal{E} \sin^2 \theta,
\] (5.30)

\[
\Phi_r(r) = \frac{a}{\Delta} \left[\mathcal{E}(r^2 + a^2) - a\mathcal{L}_z\right], \quad \Phi_\theta(\theta) = \mathcal{L}_z \csc^2 \theta.
\] (5.31)

The radial equation can be converted into one for the radial phase \( \chi^r \), and the polar equation into one for the phase \( \chi^\theta \) in a straightforward manner using (5.5) and (5.6). It is these equations that we integrate to generate Kerr orbits of fixed \((p,e,i)\) to test our frequency extraction methods.
5.8.2 Envelope subtraction method in the slow time approximation

In this appendix we demonstrate the utility of the envelope subtraction method discussed in section 5.4. The goal of this method is to locate the successive maxima $t_i^+$ or minima $t_i^-$ of the underlying motion of an oscillator with slowly varying parameters, and use them to calculate the frequency of oscillation. The variation of the amplitude and midline of the underlying oscillation shifts the position of the extrema from where they would be in the absence of the slow changes. We quantify the utility of the envelope subtraction method in removing these shifts by analyzing a toy model which captures some of the behavior of $\Omega(t)$ in our equatorial inspirals.

To model the orbital frequency, consider a function $f(\tilde{t}, t)$ of a fast time $t$ and a slow time $\tilde{t} = \epsilon t$, with $\epsilon \ll 1$ giving the ratio of inspiral to orbital time scales. In order to represent an underlying oscillation at a single fixed frequency $\omega$, we assume that $f$ is periodic with period $P = 2\pi/\omega$ in the fast time,

$$f(\tilde{t}, t + P) = f(\tilde{t}, t).$$  \hspace{1cm} (5.32)

Since $\tilde{t} = \epsilon t$, after a period $f$ does not return to precisely the same value, but is modified by the advance of the slow time variable.

The maxima of the underlying oscillation are the extrema of $f$ in the limit $\epsilon \to 0$. For simplicity we phrase the discussion in terms of maxima; the procedure is identical for minima. For nonzero $\epsilon$, we find the maxima $t_i$ by solving

$$\frac{df}{dt} = \partial_t f + \epsilon \partial_{\tilde{t}} f = 0$$ \hspace{1cm} (5.33)

order by order in $\epsilon$, under the ansatz

$$t_i = t_i^{(0)} + \epsilon t_i^{(1)} + O(\epsilon^2).$$ \hspace{1cm} (5.34)

At leading order, we have as expected

$$\partial_t f|_{t_i^{(0)}} = 0,$$ \hspace{1cm} (5.35)

which shows that $t_i^{(0)}$ are the maxima of the underlying oscillation. We seek $t_i^{(0)}$, but we have only access to $t_i$ when analyzing $f$. From equation (5.34) we see that the naive maxima $t_i$ incur an error $O(\epsilon)$. 

Returning to equation (5.33), we Taylor expand all quantities around \( t = t_i^{(0)} \) to find at the next order that
\[
t^{(1)}_i = - \frac{\partial_i f}{\partial f^2} \bigg|_{t_i^{(0)}}.
\]
(5.36)
This provides the leading error on our estimation of the maxima. We can iterate this procedure to compute higher order errors, but it turns out we can instead identify a different function, the envelope subtracted \( f \), whose maxima are closer to \( t_i^{(0)} \).

We first fit a function through the maxima \( t_i \) (in practice a spline of at least cubic order). This envelope \( f^+(\tilde{t}) \) describes the slow time variation passing from maximum to maximum, meaning that \( f^+(\tilde{t}) = f(\tilde{t}, t_i^+) \). By Taylor expansion about \( t_i^{(0)} \) we have
\[
f^+ (\tilde{t}) = f(\tilde{t}, t_i^{(0)}) - \epsilon^2 \frac{(\partial_i f)^2}{2 \partial f^2} \bigg|_{t_i^{(0)}, \tilde{t}} + O(\epsilon^3),
\]
(5.37)
where we have simplified this function by recalling that \( f(\tilde{t}, t_i^{(0)} + nP) = f(\tilde{t}, t_i^{(0)}) \).

Next, we define the “envelope-subtracted” function
\[
\hat{f} = f - f^+,
\]
(5.38)
and seek its maxima. To motivate this definition, imagine we could seek the maxima of
\[
f(\tilde{t}, t) - f(\tilde{t}, t_i^{(0)}),
\]
(5.39)
which are given by the condition
\[
\partial_t f(\tilde{t}, t) - \epsilon [\partial_t f(\tilde{t}, t) - \partial_t f(\tilde{t}, t_i^{(0)})] = 0.
\]
(5.40)
The solutions are in fact the desired maxima, \( t = t_i^{(0)} \), but we cannot formulate this ideal subtraction because we cannot identify \( f(\tilde{t}, t_i^{(0)}) \). Fortunately, \( \hat{f} \) is is equal to \( f(\tilde{t}, t_i^{(0)}) \) up to \( O(\epsilon^2) \), and so we search for the maxima of (5.38).

We find that the maxima of \( \hat{f} \), which we denote \( \hat{t}_i \), are
\[
\hat{t}_i = t_i^{(0)} + \epsilon^3 \delta t_i^{(0)} + O(\epsilon^4),
\]
(5.41)
\[
\delta t_i \equiv - \frac{(\partial_i f)^2}{\partial f^2} \partial_i \left( \ln \partial_i f - \frac{1}{2} \ln \partial^2 f \right).
\]
(5.42)
The error in \( \hat{t}_i \) (relative to the desired maxima \( t_i^{(0)} \)) is \( O(\epsilon^3) \), i.e. envelope substraction has resulted in an improvement by two powers of \( \epsilon \).
When we actually compute the frequencies, the error is even lower. We have
\[
\hat{t}_{i+1} - \hat{t}_i = \hat{t}_i^{(0)} - \hat{t}_i^{(0)} + \epsilon^3 (\delta t_{i+1} - \delta t_i) + O(\epsilon^4)
\]
\[
= P + \epsilon^3 [\delta t(\hat{t}_i^{(0)}) + P, \epsilon \hat{t}_i^{(0)} + \epsilon P] - \delta t_i + O(\epsilon^4)
\]
\[
= P + \epsilon^4 P \partial \hat{t}_i (\delta t_i)|_{\hat{t}_i^{(0)}} + O(\epsilon^5)
\]
where we have used the periodicity of \(f\) in the variable \(t\) and expanded the slow time function at \(t_{i+1}\) about \(t_i\). The above expressions are understood to be evaluated at \(t_i^{(0)}\). The differencing at the extrema leads to an error \(O(\epsilon^4)\) due to another cancellation of the leading order correction to the period, due to slow evolution of the function and its envelope from maximum to maximum. Without the envelope subtraction procedure, the differencing at successive peaks results in an \(O(\epsilon^2)\) error in the extraction of the period.

5.8.3 Evolution of eccentricity and semi-major axis

In this appendix we collect the relevant results of [239,240] for the evolution of the eccentricity of a binary at lowest PN order. In this case, we consider a binary in an eccentric, Newtonian orbit, which sources gravitational wave emission through the quadrupole formula. The resulting orbit-averaged emission of energy and angular momentum changes the Newtonian parameters over the course of the slow inspiral. Peters [239] provides differential equations for orbit-averaged changes to the semi-major axis and eccentricity, \(da/dt\) and \(de/dt\), in terms of \((a,e)\). These can be solved numerically given initial conditions \((a_0,e_0)\), or combined and integrated to give an analytic expression for \(a(e)\),
\[
a = c_0 \frac{e^{12/19}}{1 - e^2} \left(1 + \frac{121}{304} e^2\right)^{870/2299},
\]
where the integration constant \(c_0\) is determined by \((a_0,e_0)\). In order to re-express this equation as the evolution of eccentricity with averaged orbital frequency \(\langle \Omega \rangle\), we recall Kepler’s law for Newtonian orbits, \(m = \langle \Omega \rangle^2 a^3\). We can then write
\[
\frac{\langle \Omega \rangle}{\langle \Omega \rangle_0} = \left(\frac{e}{e_0}\right)^{-18/19} \left(1 - e_0^2/1 - e^2\right)^{-3/2} \left(1 + \frac{121}{304} e_0^2\right)^{-1305/2299}.
\]
Given and initial eccentricity and starting frequency, this formula provides us with an curve describing \(e(\langle \Omega \rangle)\). We compare this leading post-Newtonian result to our numerical simulations in Figure 5.5.
Figure 5.13: Performance of frequency extraction for Kerr-geodesics. Shown are the RMS differences $\delta \Omega_a^\text{rms} = \left| \Omega_a^\text{a} - \Omega_a^\text{exact} \right| / \Omega_a^\text{exact}$, where $\Omega_a^\text{a}$ is computed from the geodesic trajectory with the same techniques we employ for our BBH analysis, and $\Omega_a^\text{exact}$ is the exact frequency of the Kerr geodesic, for which we also show $\Omega^\theta$ and $\langle \Omega \rangle$. We repeat the frequency-extraction using intervals covering $N_{\text{cycles}}$ radial oscillation periods, and plot the resulting $\delta \Omega_a^\text{rms}$ as a function of $N_{\text{cycles}}$. The insets show absolute values of FFTs on $\Omega(t)$ computed over widths of 1, 3, 10 and 50 radial cycles, with the solid black line highlighting $\Omega_r^\text{exact}$. Even at 50 cycles the peaks remain unacceptably broad.

### 5.8.4 Validation of frequency extraction methods

This appendix assesses the accuracy of our frequency extraction techniques by application to Kerr geodesics, whose fundamental frequencies are known analytically. We compute one equatorial Kerr geodesic (eccentricity $e = 0.8$, semilatus rectum $p = 3.96M$), and one inclined Kerr geodesic (inclination $i = 80^\circ$, $e = 0.1$, $p = 13M$). The geodesics are integrated with high precision by an ODE integrator, but we output only a discrete time series with spacing similar to that of the BBH simulations of our main analysis. We then apply the frequency extraction techniques described in sections 5.5 and 5.6 to the discrete time series. In addition to computing frequencies over $N_{\text{cycles}} = 1$ radial cycle, we also compute these frequencies over $N_{\text{cycles}} \geq 2$ radial cycles by changing “$i + 1$” to “$i + N_{\text{cycles}}$” in equations (5.12) and (5.16). For each value of $N_{\text{cycles}}$ we compute the relative difference $\delta \Omega_i^a = (\Omega_i^a - \Omega_i^\text{exact}) / \Omega_i^\text{exact}$, and its root-mean-square

$$
\delta \Omega_i^\text{rms} \equiv \left( \frac{1}{N} \sum_i (\delta \Omega_i^a)^2 \right)^{1/2}.
$$

(5.46)

Here, $N$ is the number of elements $\delta \Omega_i^a$ for the current value of $N_{\text{cycles}}$.

The left panel of Figure 5.13 shows the results for the equatorial geodesic. Equatorial geodesics are strictly periodic and equations (5.12) and (5.16) should recover the exact
Chapter 5. Frequencies and Resonances of Generic BBH Inspirals

frequencies, up to the numerical accuracy of our extraction techniques. Indeed, \( \delta \Omega_{\text{rms}}^{\phi} \) in the left panel is dominated by our numerical procedure to compute \( \Omega \), which involves a fourth-order spline interpolant in the construction of \( \dot{r} \), and \( \delta \Omega_{\text{rms}}^{\phi} \) decreases as the fourth power of the time-sampling of the geodesic data. The accuracy of the extraction of \( \Omega_i^r \) is limited by the accuracy \( \sim 10^{-6} M_{\text{Kerr}} \) with which our procedure determines the times \( t_i^+ \) of the maxima of \( \Omega(t) \). Our extraction errors for the equatorial orbits are several orders of magnitude smaller than the systematic errors that arise due to genuine multi-periodicity for inclined orbits.

We now turn to the analysis of the inclined geodesic, which is presented in the right panel of Figure 5.13. The radial frequency \( \Omega_i^r \) is recovered to an accuracy better than 1% when \( N_{\text{cycles}} = 1 \). The polar frequency \( \Omega_i^\theta \) and the average orbital frequency \( \langle \Omega \rangle \) are recovered to nearly 0.1%. We also illustrate extraction of the azimuthal frequency \( \Omega_i^\phi \). To extract \( \Omega_i^\phi \), we begin by defining an “in-plane” separation vector \( \rho = r - (r \cdot \hat{\chi}) \hat{\chi} \), where \( \hat{\chi} \) is the direction of the black hole angular momentum. We compute an instantaneous azimuthal frequency by

\[
\Omega_i^\phi(t) = \frac{\rho \times \dot{\rho}}{\rho^2}
\]

and obtain the averaged azimuthal frequency \( \Omega_i^\phi \) by evaluating the right-hand-side of (5.12), substituting \( \Omega_i^\phi(t) \) for \( \Omega(t) \). We recover \( \Omega_i^\phi \) with a fractional accuracy of about 10% (for \( N_{\text{cycles}} = 1 \)).

The errors \( \delta \Omega_{\text{rms}}^{a} \) for the inclined geodesic stem from the interaction of the radial and polar motion. Simply put, subsequent periastron passages occur at different values of \( \theta \). As the extraction interval is lengthened (larger \( N_{\text{cycles}} \)), this dependence is averaged out over a longer time-intervals, and \( \delta \Omega_{\text{rms}}^{a} \approx 1/N_{\text{cycles}} \). The magnitudes of \( \delta \Omega_{\text{rms}}^{a} \) for each frequency depend on how important the interactions between radial and polar motion are for that particular frequency. The chosen geodesic has high inclination \( i = 80^\circ \) and fairly small eccentricity \( e = 0.1 \); while the radial motion provides the dominant modulations (and our choice to average over radial periods \([t_i^+, t_{i+N_{\text{cycles}}}^+]\) is appropriate), the considered geodesic emphasizes the impact of the polar motion. As this example demonstrates, \( \Omega_i^\phi \) is most susceptible to modulations, a finding that we confirm for the BBH systems, and which provides the basis for our preference of \( \langle \Omega \rangle \) over \( \Omega_i^\phi \).

The insets in Figure 5.13 illustrate frequency-extraction with more conventional Fourier techniques, in the form of periodograms computed from Fourier transforms of Hann-windowed samples of \( \Omega(t) \) over window-sizes of \( N_{\text{cycles}} = 1, 3, 10, \) and 50 radial passages. For sufficiently long window-sizes, the periodograms do converge to the exact Kerr-frequency. However, at window sizes practical for the inspiral rate of our BBH
Figure 5.14: Robustness of frequency-extraction under change of width of extraction window. We extract frequencies using windows of $N_{\text{cycles}}$ radial cycles, and plot the relative deviation from the same frequencies extracted at $N_{\text{cycles}} = 1$. Different panels represent different simulations, and the curves within each panel correspond to different times $\tilde{t}_i$. For the equatorial simulations (top panels), the radial and azimuthal frequencies are shown, whereas for the inclined simulations (middle and bottom panels), the radial and polar frequencies are plotted.
simulations \((N_{\text{cycles}} = 1, 3)\), periodograms do not yield any useful information, and even when using 50 radial cycles, the achieved accuracy is only \(\sim 1\%\).

Figure 5.14 examines the impact of the length of the extraction intervals (parameterized by \(N_{\text{cycles}}\)) for our BBH simulations. We extract frequencies \(\Omega_{a}^N\) computed over intervals of length \(N = N_{\text{cycles}}\), which we associate with the midpoint times \(\tilde{t}_{i,N}^+ = (t_{i}^+ + t_{i+N}^+)/2\). We interpolate the time series \(\Omega_{a}^N\) onto the mid-times \(\tilde{t}_{i}^+\), and compute the relative difference from the \(N_{\text{cycles}} = 1\) case,

\[
\Delta \Omega_{a}^N = \frac{\Omega_{N}^a - \Omega_{1}^a}{\Omega_{1}^a}, \tag{5.48}
\]

where the subscript denotes the value of \(N_{\text{cycles}}\) over which the frequency is computed. Because the characteristic frequencies increase during the inspiral, we report \(\Delta \Omega_{N}^a\) at select values of \(\Omega_{a}\), rather than the RMS error across the whole series.

In all cases we find that \(\Delta \Omega_{a}\) increases with \(N_{\text{cycles}}\), with the largest \(\Delta \Omega_{a}\) closer to merger (i.e. at larger \(\tilde{t}_{i}^+\)). For equatorial orbits our frequency extraction is very precise, and so \(N_{\text{cycles}} = 1\) will ensure the least impact by errors arising from dissipative inspiral. For precessing orbits, there are two competing effects: First, with increasing \(N_{\text{cycles}}\), the impact of cycle-to-cycle modulations diminishes, leading to extracted sequences \(\Omega_{i}^a\) that show less variations between neighboring extrema \(i\). Second, increasing \(N_{\text{cycles}}\) leads to larger systematic biases of the extracted frequencies \(\Omega_{i}^a\), cf. Figure 5.14. To avoid contamination by such systematic biases, we choose \(N_{\text{cycles}} = 1\) throughout, accepting possibly larger variations of the extracted frequencies.

Some of our simulations result in extracted frequencies that are noticeably more erratic than the rest of the runs. This effect is most striking for the low-eccentricity highly-inclined orbits, cf. lower left panel of Figure 5.9. These simulations are ill-behaved throughout our study, yielding far less smooth precession rate curves than their counterparts (cf. Figure 5.9). Study of the separation \(r(t)\) for these simulations (cf. Figure 5.2) reveals unusually strong aperiodic features in the trajectories, which persists across our different numerical resolutions, which we show in Figure 5.15. Curiously, the aperiodicity is far less dramatic at higher eccentricities, and indeed worsens during approach to merger, where the eccentricity is lower. This may be a sign that for these particular parameter choices quasi-periodic effects are comparable to dissipative ones, and that therefore \(N_{\text{cycles}} = 1\) (or any constant \(N_{\text{orbits}}\)) is inappropriate here. Alternatively, this may be an effect of the damped-harmonic SpEC simulation gauge [86, 198, 292], whose behaviour is largely untested for dynamically generic orbits.
Section 5.8.5 Estimation of resonant accretion

To increase the sensitivity to potential resonant effects, we can average the fluxes $\dot{J}_\perp$, $\dot{J}_\parallel$ and $\dot{E}$ over $N$ radial oscillation periods, where we slide the averaging window to start at each maximum of the respective flux. From (5.22), we expect that non-resonant terms are averaged out by this procedure. Therefore, off-resonance, the averaged fluxes

$$\langle \dot{C} \rangle \approx \dot{C}_{00}$$

(5.49)

correspond to the $\dot{C}_{00}$ term. When averaging over a time window smaller than the time on resonance, and when the window is centered on the resonance, we can approximate $\Phi_{kn}(t) \approx \Phi_{kn}(t_{\text{res}})$ over the averaging window. Then (5.22) yields

$$\langle \dot{C} \rangle \approx \dot{C}_{00} + \dot{C}_{kn} e^{-i \Phi_{kn}(t_{\text{res}})/\dot{C}_{00}}.$$ 

(5.50)

Therefore, during resonance, the averaged fluxes $\langle \dot{C} \rangle$ should show a relative deviation with peak amplitude $\delta \dot{C} \sim \dot{C}_{kn} e^{-i \Phi_{kn}(t_{\text{res}})/\dot{C}_{00}}$ compared to orbits off-resonance.

The resulting averaged fluxes are shown in the top panel of Figure 5.16 for two of our most extreme simulations, $q7_i40\_high-e$ and $q7_i80\_high-e$. On the vertical scale of these plots, no variations of the fluxes are discernible between on- and off-resonance. To sharpen our bound, we remove the overall inspiral component of the fluxes through a fit to a function $f_{\text{fit}}(t) = A_0(A_1 - t)^{A_2} + A_3$. The functional form is chosen to capture the overall inspiral behavior, while not being able to capture intermediate variations that are
expected on resonance. Therefore, we expect \( f_{\text{fit}}(t) \approx \dot{\mathcal{C}}_{00} \). The normalized residual

\[
R(t) = \frac{\langle \dot{\mathcal{C}} \rangle - f_{\text{fit}}}{\langle \dot{\mathcal{C}} \rangle}
\]  

(5.51)

should therefore vanish off-resonance. On-resonance, (5.50) suggests that (dropping terms of order unity) \( R \sim \dot{\mathcal{C}}_{kn} e^{-i\Phi_{kn}(t_{\text{res}})}/\dot{\mathcal{C}}_{00} \). As can be seen from the lower panels of Figure 5.16, we find numerically \( R \lesssim 0.004 \). The short-period variations in \( R \) in this Figure are caused by the numerical accuracy with which we can extract periastron passages and perform the averaging. The overall smooth trend (\( R \) is slightly negative at small \( t \)) indicates the quality with which our fitting function \( f_{\text{fit}} \) can capture the overall inspiral dynamics. Besides these two properties of \( R(t) \), Figure 5.16 does not show any systematic deviations at the highlighted resonances. Therefore, we conclude for these resonances that

\[
\frac{\dot{\mathcal{C}}_{kn} e^{-i\Phi_{kn}(t_{\text{res}})}}{\dot{\mathcal{C}}_{00}} \lesssim 0.004.
\]  

(5.52)
Chapter 6

Conclusions

In this thesis we have presented projects designed to improve the wall-clock time, ease of development, and ease of interpretation of numerical relativity simulations. Chapter 3 presented our GPU port of the numerical relativity code SpEC, illustrating its maintainability and performance. Chapter 4 focussed on part of this port, TLoops, which represents tensor manipulations in C++ code using analytically-familiar notation, and which generates C or CUDA code automatically. Chapter 5 presented a study of eccentric black hole binaries simulated with SpEC, and developed numerical techniques for extracting their precession rates. Using those precession rates, we were able to show that our simulated binaries pass through interesting coordinate resonances, but that these have no measurable effect on the inspiral.

Our GPU port used a combination of explicit kernels, partially-automated porting by tracing matrix representations of existing functions, and fully-automated porting by generating code from expression templates. These choices were made to balance maintainability and performance. We demonstrated efficient GPU use across all the cards we tested on and achieved a 5-10X overall speedup vs. a CPU configured in a way SpEC would realistically use.

Future work on the GPU port will focus on producing production-quality binary black hole simulations. These involve a few more modules that detect apparent horizons and interpolate quantities onto them. Possible interactions between the GPU port, mesh refinement, and MPI load balancing will also need to be considered.

TLoops furnishes an expression-template based framework allowing tensor-calculus operations to be written directly into C++ source code. Those operations can subsequently be used to generate equivalent CUDA or low-level C++ code, which can then be linked back to the original project. In Chapter 4 we outlined the hierarchy of templates used to create this library and to make possible its automatic code-generation capabi-
ties. We then presented benchmarks demonstrating our library’s efficiency, particularly in the GPU case.

Future steps regarding TLoops will begin by updating the Tensor class used by SpEC to represent mathematical tensors such that individual element arrays are stored contiguously. This will make possible considerable simplifications and speedups throughout the library. We furthermore intend to publish TLoops separately from SpEC, such that it can be used in concert with other projects.

Finally, we performed a suite of long BBH simulations with high spins, moderate eccentricities, and moderate mass-ratio. We developed numerical techniques to extract fundamental frequencies from these simulations despite their multi-frequency behaviour, and used these to compute gauge-insensitive precession rates. These allowed us to make direct comparisons with analytic approximation theory, and to demonstrate that our simulation passed through interesting orbital resonances. We showed that these resonances had no effect on the inspirals.

Future work here will attempt to push to higher mass-ratios and eccentricities. Simulations of suites of inspirals with near-constant intrinsic parameters will allow us to better detect phase-dependent (and thus resonant) behaviour.

Our present focus, however, is on the use of numerical relativity to dynamically evolve quantum fields. Since about the mid-80s, condensed matter theorists (mostly) have developed a set of numerical techniques known collectively as “tensor networks”. As is well known, the (e.g.) space required to numerically represent a general quantum state scales exponentially with the “size” of the system. Taking for example a spin chain, tensor networks relegate that exponential scaling to the system’s degree of entanglement rather than the number of lattice sites. Since states near the ground state for a wide class of Hamiltonians often exhibit sublinear scaling of entanglement (in a certain technical sense) with energy, this permits an efficient representation of practical systems.

These techniques have over the past few decades been extended to two and three dimensional systems, to conformal field theories, and to quantum field theories more generally. Unlike for example quantum Monte Carlo methods, their ability to explicitly represent the state makes them useful for time-dependent problems. These arise, for example, in curved spacetime, where the Hamiltonian is a function of the 3+1 variables.

This presents a number of interesting opportunities. Most obviously, we are currently developing means to compute e.g. field correlators in general curved backgrounds, allowing for investigation of nonlinear gravitational effects upon fields that include strongly-coupled interaction terms, upon spacetimes with complex dynamics. Such problems, which aside from their foundational interest are relevant to for example inflationary cos-
mology, are analytically very challenging in such cases. Using the correlators the field’s expected stress-energy could also be computed numerically at each timestep. This could then be fed into a numerical relativity situation which would use it as a matter source, allowing so called “semiclassical gravity” effects to be computed. Slightly adjacently, the “degree of entanglement” inherent to a system, which dictates the complexity of the numerical representation, is not a gauge (or even a Lorentz) invariant. We are currently investigating the possibility of finding optimized slicing choices. This could conceivably result in new methods for time-dependent simulations of field theories in Minkowski space as well.

We thank the reader for their participation.
Bibliography


[7] B. P. Abbott et al. GW150914: First results from the search for binary black hole coalescence with Advanced LIGO. 2016. submitted to to PRD.


[175] Sebastian Khan, Sascha Husa, Mark Hannam, Frank Ohme, Michael Pürrer, Xisco Jiménez Forteza, and Alejandro Bohé. Frequency-domain gravitational waves from non-precessing black-hole binaries. II. A phenomenological model for the advanced detector era. 2015.


[224] NVIDIA. Fermi tuning guide.


[318] Carroll L Wainwright, Matthew C Johnson, Hiranya V Peiris, Anthony Aguirre, Luis Lehner, and Steven L Liebling. Simulating the universe (s): from cosmic


