Parallelization And Scalability Analysis Of The \[1pc\] 3D Spatially Variant Lattice Algorithm

Henry Roger Moncada Lopez

University of Texas at El Paso, hrmoncada@gmail.com

Follow this and additional works at: https://digitalcommons.utep.edu/open_etd

Part of the Computer Sciences Commons, Electrical and Electronics Commons, and the Electromagnetics and Photonics Commons

Recommended Citation

https://digitalcommons.utep.edu/open_etd/1492

This is brought to you for free and open access by DigitalCommons@UTEP. It has been accepted for inclusion in Open Access Theses & Dissertations by an authorized administrator of DigitalCommons@UTEP. For more information, please contact lweber@utep.edu.
PARALLELIZATION AND SCALABILITY ANALYSIS OF THE
3D SPATIALLY VARIANT LATTICE ALGORITHM

HENRY R. MONCADA LOPEZ

DOCTORAL PROGRAM IN COMPUTATIONAL SCIENCE

APPROVED:

Shirley V. Moore, Ph.D., Chair

Raymond C. Rumpf, Ph.D.

Eric Freudenthal, Ph.D.

Natasha S. Sharma, Ph.D.

Charles Ambler, Ph.D.
Dean of the Graduate School
©Copyright

by

Henry R. Moncada Lopez

2018
To my son Michael D. Moncada, my mother, brothers and sister
with Love
WHAT I LOOK AT EACH DAY?

“There are a few things that important in life, about these. Three things to my count, are the ones I need each day. One of them is something to look up to, another is something to look forward to, and another is someone to chase for.

**Something to look up to**, I want to thank God because that’s whom I look up to. He has graced my life with opportunities that I know aren’t of my hand or any other human hand. He has shown me that it’s a scientific fact that gratitude reciprocates. In the words of the late Charlie Laughton, who said, When you got God, you got a friend and that friend is you.

**Something to look forward to**, to my family, that’s who and what I look forward to. To my father, I know he’s up there right now probably with a cold beer on his hand and he is dancing right now. To you dad, you taught me what it means to be a man, to assume my own responsibility, and who always came with a solution in mind when something is wrong. I need to report but also to solve. To my mother, who taught my two brothers, sister and me, demanded that we respect ourselves and what we in turn learned was then we were better able to respect others. Thank you for that, Mama. To my son Michael, the courage and significance you give me every day I go out the door to front the facts of life and see if I could not learn what I have to teach, and not, are unparalleled. To the forgotten girl that stole my heart, I want it back!!!...please!!! You’re indeed my sweet hard vanilla cookie. I learned with you, that love doesn’t care who knows more and that time doesn’t exist in us for all that I see in you. You’re my best friend!. To my friends, You!! guys are the people that look at me, in my best and worst moments in life. You guys show me how to laugh at myself and show sympathy, tolerance, and kindness to others. To all of you!! Thank you!! You are the people in my life that I want to make the most proud of me.

**Someone to chase for**, to my hero, that’s who I chase. Now, when I was 15 years old, I had a very important person in my life come to me and say, Who’s your hero? and I said, I don’t know, I’ve got to think about that. Give me a couple of weeks. I come back two weeks later, this person comes up and says, Who’s your hero? I said, I thought about it. It’s me in 10 years. So I turned 25. Ten years later, that same person comes to me and says, So, are you a hero? And I was like, Not even close! No, no no! Did she say, Why? I said because my hero’s me at 35. So you see every day, every week, every month, and every year of my life, my hero’s always ten years away. I’m never going to be my hero. I’m not going to attain that. I know I’m not, and that’s just fine with me because that keeps me with somebody to keep on chasing.

So, to any of us, whatever those things are, whatever it is we look up to, whatever it is we look forward to, and whoever it is we’re chasing, keep living, keep fighting, keep pushing forward, don’t waste your time regretting things you didn’t do well, learn from your mistakes, made the best from everything you learn, and remember!!! happiness is a direction, not a destination. Your life is a continuous balancing act that has less to do with pain and more to do with beauty.”

Matthew McConaughey Oscar speach 2014, Henry D. Thoreau, and Shane Koyczan
PARALLELIZATION AND SCALABILITY ANALYSIS OF THE
3D SPATIALLY VARIANT LATTICE ALGORITHM

by

HENRY R. MONCADA LOPEZ

THESIS
Presented to the Faculty of the Graduate School of
The University of Texas at El Paso
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

DOCTORAL PROGRAM IN COMPUTATIONAL SCIENCE
THE UNIVERSITY OF TEXAS AT EL PASO
May 2018
Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Shirley Moore of the Computer Science Department at The University of Texas at El Paso, for the continuous support of my study and related research, for her patience, motivation, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Raymond Rumpf of the Electrical & Computer Engineering Department, Prof. Eric Freudenthal Computer Science Department, and Dr. Natasha Sharma of the Mathematics Department, all at The University of Texas at El Paso, for their insightful comments and encouragement, but also for the hard questions that give me the incentive to widen my research from various perspectives.

I thank my dear fellow classmates for the stimulating discussions, loving support and no complaint and for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last years. Last but not the least, I would like to thank my family: my parents and my brothers and sister for supporting me spiritually throughout writing this thesis and my life in general.

The authors acknowledge the Air Force Office of Scientific Research and Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing resources that have contributed to the research results reported in this paper (URL: http://www.tacc.utexas.edu).

NOTE: This thesis was submitted to my Supervising Committee on May 4, 2018.
Abstract

The purpose of this research is to design a faster implementation of an algorithm to generate 3D spatially variant lattices (SVL) and improve its performance when it is running on a parallel computer system.

The algorithm is used to synthesize a SVL for a periodic structure. The algorithm has the ability to spatially vary the unit cell, the orientation of the unit cells, lattice spacing, fill fraction, material composition, and lattice symmetry. The algorithm produces a lattice that is smooth, continuous and free of defects. The lattice spacing remains strikingly uniform even when the lattice is spatially varied. This is important for maintaining consistent properties throughout the lattice. Periodic structures like photonic crystals or metamaterial devices can be enhanced using the spatially variant algorithm, thus unlocking new physical mechanisms.

Our current effort is to write a portable computer program for parallel architectures. To develop and write the code, we pick a general purpose programming language that supports structured programming. For the parallel code, we use the FFTW (Fastest Fourier Transform in the West) for handling the Fourier transform of the unit cell device and PETSc (Portable, Extensible Toolkit for Scientific Computation) for handling the numerical linear algebra operations. Using Message Passing Interface (MPI) for distributed memory helps us to improve the performance of the code that generates 3D SVL when it is executed on a parallel system.

We study the efficiency and scalability of the SVL code on the two architectures on Stampede2 (KNL and SKX). A scalable code can handle proportionally very small to large tasks of computational operations. To scale the SVL code, we use isoefficiency analysis to proportionally increase the problem size and resources to study the scalability of the SVL code. This type of analysis is important to understand the performance of the SVL code when it is executed on a supercomputer and find the best way to maximize
its performance.

Our future work consists of improving the efficiency and scalability by reducing the overload performance of communication overhead. Moving data limited the performance achievement of the SVL code. Improving the scalability as the problem size and the number of processors grow is an essential factor in getting a better performance of the SVL code.
# Table of Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>vii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>ix</td>
</tr>
</tbody>
</table>

## Chapter 1 Background on the Technique

1.1 Periodic Structures                                 | 1      |
1.2 The Finite-Difference Method                        | 2      |
1.2.1 Taylor series                                    | 3      |
1.2.2 Approximation of the first-order derivative       | 4      |
1.2.3 Approximation of the second-order derivative      | 6      |
1.2.4 Grid Points                                       | 7      |
1.2.5 Boundary Conditions for 1D                        | 8      |
1.3 Finite-Difference Derivative Operator               | 11     |
1.3.1 Dirichlet Boundary Conditions                     | 12     |
1.3.2 Periodic Boundary Conditions                      | 12     |
1.3.3 Neumann Boundary Conditions                       | 13     |
1.4 Linear Algebra                                      | 17     |
1.4.1 Overdetermined system                             | 17     |
1.4.2 Least Squares                                     | 17     |
1.5 Electromagnetic Fields                              | 18     |
1.6 Spatially Varying a Planar Grating                  | 19     |
1.6.1 Solving the Gradient Equation                     | 21     |
1.7 Why the Grating phase $\Phi(\vec{r}) = \vec{K} \cdot \vec{r}$ fails | 22     |
1.8 Spatially Varianing Grating Examples                 | 24     |
2 3D Spatially Variant Lattice Program Code Implementation 26
   2.1 Motivation 26
   2.2 SVL 3D Program Code 27
      2.2.1 Unit Cell Construction 27
      2.2.2 Fourier Transform 28
      2.2.3 Swap Quadrants 32
      2.2.4 Truncate Spatial Harmonics 32
      2.2.5 Grating Vector 33
      2.2.6 Device Parameters 34
      2.2.7 Spatially Varying the Lattice 36
      2.2.8 Orientation Attribute 37
      2.2.9 Lattice Spacing 40
      2.2.10 Finite-Difference Derivative Operators (FDDER) for 3D 40
      2.2.11 SVL Loop 45
      2.2.12 Form Binary Grating Using Spatially Variant Threshold 50
   2.3 SVL Pseudocode 51
      2.3.1 Algorithm 1: Main Program 54
      2.3.2 Algorithm 2: Unit Cell 55
      2.3.3 Algorithm 3: Implementation of the Fourier Transform 57
      2.3.4 Algorithm 5: Loop to Solve the Linear System 61
3 Improving the Efficiency of the SVL Algorithm 64
   3.1 Motivation 64
   3.2 Bottlenecks Description 65
      3.2.1 Number of Harmonics 65
      3.2.2 Direct or Iterative Method 65
   3.3 Improvements Implemented 67
      3.3.1 Elimination of the Gratings According to Their Amplitude 68
4 Results and Analysis 71
Chapter 1

Background on the Technique

In this chapter, a compelling method for bending, twisting, and otherwise spatially varying periodic structures is presented. The technique is a numerical method that allows us to manipulate any of the attributes of a periodic structure (lattice properties). Here we present the ideas behind the technique that will allow an understanding of our implementation. The technique is used to functionally grade the lattice structure without changing the size and shape of the unit cells, keeping the lattice structure smooth and continuous. This technique allows us to minimize deformations and prevent defects being produced in the final lattice. Since it is not a coordinate transformation, it can more easily produce complicated and arbitrary spatial variance structures [19].

1.1 Periodic Structures

Periodic structures are repeated geometry structures, composed of small repeated entities called “unit cells”. The unit cell structure can be superimposed on itself by a parallel displacement over a certain finite distance. Each unit cell corresponds to one structural period and depending on the number of independent directions of displacement, we can construct periodic structures in one, two, and three dimensions.

For example, in 1D the minimum distance value $T$ is called the period. In theory, periodic structures are infinite and serve as idealized models for the theoretical study of real lattices. In practice, a structure is considered periodic if it can be described by only one repeated element (unit cell) that is connected to identical elements in an identical manner. As in actual fabrication, periodic structures do not remain exactly geometri-
cally uniform over a structural period due to inherent imperfections involved in the fabrication process as shown in Figure 1.2.

Figure 1.1: (a) Primitive 3D unit cell and (b) 3D periodic structure[12]

Consider a 1D periodic structure in $x$. This 1D structure is infinite in $\pm x$ with a period $T$. Let us denote this by a periodic function $P(x)$ that represents a field reaction of the 1D periodic structure. Now, let us consider a periodic function $P(x)$ of the form

$$P(x + nT) = P(x)$$

Thus, $P(x)$ is a periodic function in $x$, with period $T$ that can be expressed via a Fourier Series expansion. Notice that the periodic phase shift and attenuation are normalized out in $P(x)$.

$$P(x) = \sum_{n=-\infty}^{\infty} p_n e^{j2\pi nx/T} \quad (1.1)$$

### 1.2 The Finite-Difference Method

The finite-difference method is a numerical technique to solve differential equations. This technique handles each derivative in the equation by it replacing with the finite-
difference approximation. A differential quotient is built by using differences in the values of the function between a given value of the independent variable, say $x$, and a small increment $(x + h)$. For example, let us consider the 1D case only. The derivative $u'(x)$ can be approximated by using the finite-difference approximation with a small value of $h$ at a point $x \in \mathbb{R}$:

$$
\frac{d}{dx} u(x) \approx \lim_{h \to 0} \frac{u(x + h) - u(x)}{h}
$$

And, when $h$ tends to 0 (without vanishing), the quotient on the right-hand side provides a good approximation of the derivative. In other words, $h$ should be sufficiently small to get a good approximation.

### 1.2.1 Taylor series

Suppose the function $u(x)$ is $C^p$ continuous in the neighborhood of $x$. For any $h > 0$ we have:

$$
u(x + h) = u(x) + \frac{h}{1!}u'(x) + \frac{h^2}{2!}u''(x) + \frac{h^3}{3!}u'''(x) \cdots$$

$$
u(x - h) = u(x) - \frac{h}{1!}u'(x) + \frac{h^2}{2!}u''(x) - \frac{h^3}{3!}u'''(x) \cdots$$

The error between the numerical solution and the exact solution is determined by the error that is generated by going from a differential operator to a difference operator. This error is called the discretization error or truncation error. The term truncation error reflects the fact that a finite part of a Taylor series is used in the approximation. For the sake of convenience, let us retain only the first two terms of the previous expression:

$$
u(x + h) = u(x) + hu'(x) + O(h^2)$$

$$
u(x - h) = u(x) - hu'(x) + O(h^2)$$

where the term $O(h^2)$ indicates that the error of the approximation is proportional to $h^2$. 
1.2.2 Approximation of the first-order derivative

Definition 1. The approximation of the derivative of \( u(x) \) at point \( x \) is of order \( p \), \( p > 0 \) if there exists a constant \( C > 0 \), independent of \( h \), such that the error between the derivative and its approximation is bounded by \( Ch^p \) (i.e., is exactly \( O(h^p) \)). Assuming equal mesh spacing \( h = x_{i+1} - x_i \):

\[
\begin{align*}
x_i - 1 & \quad x_i & \quad x_{i+1} & \quad x_{i+2} & \quad x_{i+3} \\
\bullet & \quad \bullet & \quad \bullet & \quad \bullet & \quad \bullet \\
\end{align*}
\]

Figure 1.3: 1D Grid in \( x \)

Forward Difference Approximant of \( u \) at point \( x \)

\[
\begin{align*}
u(x + h) &= u(x) + hu'(x) + O(h^2) \\
u'(x) &= \frac{u(x+h)-u(x)}{h} + O(h)
\end{align*}
\]

(1.2)

From the equation (1.2), there exists a constant \( C > 0 \), such that for \( h \in (0,h_0) \), \( h > 0 \) sufficiently small, we have the following bound on the approximation error:

\[
\left| \frac{u(x + h) - u(x)}{h} - u'(x) \right| \leq Ch, \quad C = \sup_{y \in [x,x+h_0]} \frac{|u''(y)|}{2},
\]

for \( h \leq h_0 \) (\( h_0 > 0 \) given). The error generated by replacing the derivative \( u'(x) \) by the differential quotient is of order \( h \). The approximation of \( u \) at point \( x \) is said to be consistent at the first order.
Backward Difference Approximant of $u$ at point $x$

$$u(x - h) = u(x) - hu'(x) + O(h^2)$$

$$u'(x) = \frac{u(x) - u(x-h)}{h} + O(h) \tag{1.3}$$

The bound error approximation generated by replacing the derivative $u'(x)$ by the differential quotient is of order $h$, similar to the forward approximation. Other approximants can be considered to improve the accuracy of the approximation.

Central Difference Approximant of $u$ at point $x$

We define a consistent approximation by taking the points $x - h$ and $x + h$ into account.

Suppose that the function $u$ is three times differentiable in the vicinity of $x$:

$$u(x + h) = u(x) + hu'(x) + \frac{h^2}{2}u''(x) + \frac{h^3}{6}u^{(3)}(x) + O(h^4)$$

$$u(x - h) = u(x) - hu'(x) + \frac{h^2}{2}u''(x) - \frac{h^3}{6}u^{(3)}(x) + O(h^4)$$

By subtracting the two expressions we obtain the following, thanks to the intermediate value theorem:

$$\frac{u(x + h) - u(x - h)}{2h} = u'(x) + \frac{h^2}{6}u^{(3)}(x) + O(h^4)$$

$$\frac{u(x + h) - u(x - h)}{2h} = u'(x) + O(h^2)$$

where the term $O(h^2)$ indicates that the error of the approximation is proportional to $h^2$. There exists a constant $C > 0$ such that for $h \in (0, h_0)$, $h > 0$ sufficiently small, we have the following bound on the approximation error:

$$\left| \frac{u(x + h) - u(x - h)}{2h} - u'(x) \right| = Ch^2, \quad C = \sup_{y \in [x-h_0, x+h_0]} \left| \frac{u^{(3)}(y)}{6} \right|$$
### 1.2.3 Approximation of the second-order derivative

Suppose $u$ is a $C^4$ continuous function on an interval $[x - h_0, x + h_0]$, $h_0 > 0$. Then, there exists a constant $C > 0$ such that for every $h \in (0, h_0)$ we have:

$$\left| \frac{u(x + h) - 2u(x) + u(x - h)}{h^2} - u''(x) \right| = Ch^2$$

The differential quotient

$$\frac{u(x + h) - 2u(x) + u(x - h)}{h^2}$$

is a consistent second-order approximation of the second derivative of $u$ at point $x$. The approximation of the second derivative is:

$$u(x + h) = u(x) + hu' + \frac{h^2}{2}u'' + \frac{h^3}{6}u^{(3)} + \frac{h^4}{12}u^{(4)}$$

$$u(x - h) = u(x) - hu' + \frac{h^2}{2}u'' - \frac{h^3}{6}u^{(3)} + \frac{h^4}{12}u^{(4)}$$

By adding these two expressions we obtain the following, thanks to the intermediate value theorem:

$$u(x + h) + u(x - h) = 2u + \frac{2h^2}{2}u'' + \frac{2h^4}{12}u^{(4)} + O(h^6)$$

$$\frac{u(x + h) - 2u(x) + u(x - h)}{h^2} = u'' + \frac{h^2}{6}u^{(4)} + O(h^2)$$

where the term $O(h^2)$ indicates that the error of the approximation is proportional to $h^2$.

There exists a constant $C > 0$, such that for $h \in (0, h_0)$, $h > 0$ sufficiently small, we have the following bound on the approximation error:

$$\left| \frac{u(x + h) - 2u(x) + u(x - h)}{h^2} - u''(x) \right| = Ch^2, \quad C = \sup_{y \in [x-h_0, x+h_0]} \frac{|u^{(4)}(y)|}{12}$$
1.2.4 Grid Points

To find a numerical solution with finite-difference methods, we first need to define a set of grid points in the domain $D$ as follows. Here we are using $\Delta x$ and $\Delta y$ instead of $h$, for the purpose of explaining the step size in each dimension.

**For 1D:** the domain is defined as $D = [a, b]$, and we choose a step size $\Delta x$, as follows:

$$\Delta x = \frac{b - a}{N} \quad (N \text{ is an integer})$$

As shown in Figure 1.4, the horizontal intersection points $x_j$ can be drawn across $D$, using

$$x_j = a + j\Delta x, \quad j = 0, \cdots, N$$

![Figure 1.4: 1D Grid in $x$](image)

**For 2D:** the domain is defined as $D = [a, b] \times [c, d]$, and we choose a step size $\Delta x$ as follows:

$$\Delta x = \frac{b - a}{M} \quad ; \quad \Delta y = \frac{d - c}{N} \quad (M, N \text{ are integer})$$

As shown in Figure 1.5, a set of horizontal and vertical lines can be drawn across $D$, using all intersection points $(x_i, y_j)$, or simply $(i, j)$, using

$$x_i = a + i\Delta x, \quad i = 0, \cdots, M,$$

$$y_j = c + j\Delta y, \quad j = 0, \cdots, N,$$
1.2.5 Boundary Conditions for 1D

Consider the simple ODE form (i.e., 1D form) with $x \in [0, 1]$.

\[
\frac{d}{dx} u = f(x) \\
\frac{d^2}{dx^2} u = g(x)
\]

The equation is usually complemented with one of three common boundary conditions:

1. Periodic boundary condition, where $u(a) = u(b)$, equal function values outside on the opposite sides of the grid.

2. Dirichlet boundary condition, where $u(a) = 0$ and $u(b) = 0$, function values outside of the grid are zero

3. Neumann boundary condition, where $u'(a)$ = continuous and $u'(b)$ = continuous, allows functions values to continue linearly off of the grid as if to infinity.
In the following section we will discretize the interval of $x$ into $N$ points, thereby defining the discretized $x_j$:

$$x_j = a + jh, \quad h = \frac{b - a}{N}, \quad j = 0, \ldots, N$$

Let us define $u_j \approx u(x_j)$ in order to approximate the solution, where:

$$\frac{du}{dx} \approx \frac{u_{j+1} - u_{j-1}}{2h}$$

$$\frac{d^2u}{dx^2} = \left[ \frac{d}{dx} \left( \frac{du}{dx} \right) \right] = \lim_{h \to 0} \frac{\left( \frac{du}{dx} \right)_{j+1/2} - \left( \frac{du}{dx} \right)_{j-1/2}}{h} \approx \frac{u_{j+1} - u_j}{h} - \frac{u_j - u_{j-1}}{h} \approx \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2}$$

Note that the local truncation error (LTE) of these approximations is $O(h)$ for the first order derivative and $O(h^2)$ for the second order derivative (See sections 1.2.2, 1.2.3).

Now let us build a matrix operator for every boundary condition. First we explain the meaning of the superscript and subscript notations. For example, the superscript $P$ on $D^P_x$ emphasizes that this discretized matrix operator is specific only for periodic problems, $D$ on $D^D_x$ is for Dirichlet problems, and $N$ on $D^N_x$ is for Neumann problems. The subscript $x$ on $D_x$ emphasizes that this is a first-order derivative and $xx$ on $D_{xx}$ is a second-order derivative.

**Periodic Boundary Conditions**

The matrix operator $D^P_x$ of the first-order derivative with periodic boundary condition $u(0) = u(N)$ is substituted, giving the following discretized matrix for the nodes $j = 0, \ldots, N$
\[
\frac{\partial u_0}{\partial x} \approx \frac{u_1 - u_3}{2\Delta x} \\
\frac{\partial u_1}{\partial x} \approx \frac{u_2 - u_0}{2\Delta x} \\
\frac{\partial u_2}{\partial x} \approx \frac{u_3 - u_1}{2\Delta x} \\
\frac{\partial u_3}{\partial x} \approx \frac{u_0 - u_2}{2\Delta x}
\]

\[
\Rightarrow \frac{1}{2\Delta x} \begin{bmatrix}
0 & 1 & 0 & -1 \\
-1 & 0 & 1 & 0 \\
0 & -1 & 0 & 1 \\
1 & 0 & -1 & 0
\end{bmatrix} \begin{bmatrix}
u_0 \\
u_1 \\
u_2 \\
u_3
\end{bmatrix}
\]

**Dirichlet Boundary Conditions**

The matrix operator \( D_x^D \) of the first-order derivative with Dirichlet boundary conditions \( u(0) = 0 \) and \( u(N) = 0 \) is substituted, giving the following discretized matrix for the nodes \( j = 0, \ldots, N \):

\[
\frac{\partial u_0}{\partial x} \approx \frac{u_1 - u_3}{2\Delta x} \\
\frac{\partial u_1}{\partial x} \approx \frac{u_2 - u_0}{2\Delta x} \\
\frac{\partial u_2}{\partial x} \approx \frac{u_3 - u_1}{2\Delta x} \\
\frac{\partial u_3}{\partial x} \approx \frac{0 - u_2}{2\Delta x}
\]

\[
\Rightarrow \frac{1}{2\Delta x} \begin{bmatrix}
0 & 1 & -1 \\
-1 & \ddots & \ddots \\
\ddots & \ddots & \ddots & 1 \\
1 & -1 & 0
\end{bmatrix} \begin{bmatrix}
u_0 \\
u_1 \\
u_2 \\
u_3
\end{bmatrix}
\]

**Neumann Boundary conditions**

The matrix operator \( D_x^N \) of the first-order derivative with Neumann boundary conditions (See equations 1.4) is substituted, giving the following discretized matrix for the
nodes $j = 0, \ldots, N$:

$$u'_0(x) \approx \frac{u_1-u_0}{\Delta x}$$

$$u'_N(x) \approx \frac{u_N-u_{N-1}}{\Delta x}$$

$$\begin{align*}
\frac{\partial u_0}{\partial x} & \approx \frac{u_1-u_0}{\Delta x} \\
\frac{\partial u_1}{\partial x} & \approx \frac{u_2-u_0}{2\Delta x} \\
\frac{\partial u_2}{\partial x} & \approx \frac{u_3-u_1}{2\Delta x} \\
\frac{\partial u_2}{\partial x} & \approx \frac{u_3-u_2}{\Delta x}
\end{align*}$$

$$D_N^x = \frac{1}{2\Delta x} \begin{bmatrix}
-2 & 2 & 0 & 0 \\
-1 & 0 & 1 & 0 \\
0 & -1 & 0 & 1 \\
0 & 0 & -2 & 2
\end{bmatrix}$$

### 1.3 Finite-Difference Derivative Operator

Finite-difference approximations on a 2D grid are as follows:

$$\frac{\partial \Phi(x_i, y_j)}{\partial x} \approx \frac{\Phi(x_{i+1}, y_j) - \Phi(x_{i-1}, y_j)}{2\Delta x}$$

$$\frac{\partial \Phi(x_i, y_j)}{\partial y} \approx \frac{\Phi(x_i, y_{j+1}) - \Phi(x_i, y_{j-1})}{2\Delta y}$$

Using these finite-difference equations, we can construct the derivative matrices.
1.3.1 Dirichlet Boundary Conditions

Assume function values from outside of the grid are zero (meaning the boundary is isolated):

\[
\begin{align*}
\frac{\partial \Phi(x_1, y_j)}{\partial x} & \approx \frac{\Phi(x_2, y_j) - 0}{2\Delta x} \\
\frac{\partial \Phi(x_{N_x}, y_j)}{\partial x} & \approx \frac{0 - \Phi(x_{N_x-1}, y_j)}{2\Delta x} \\
\frac{\partial \Phi(x_i, y_1)}{\partial y} & \approx \frac{\Phi(x_i, y_2) - 0}{2\Delta y} \\
\frac{\partial \Phi(x_i, y_{N_y})}{\partial y} & \approx \frac{0 - \Phi(x_i, y_{N_y-1})}{2\Delta y} \\
\frac{\partial^2 \Phi(x_1, y_j)}{\partial x^2} & \approx \frac{\Phi(x_2, y_j) - 2\Phi(x_1, y_j) + 0}{(\Delta x)^2} \\
\frac{\partial^2 \Phi(x_{N_x}, y_j)}{\partial x^2} & \approx \frac{0 - 2\Phi(x_{N_x}, y_j) + \Phi(x_{N_x-1}, y_j)}{(\Delta x)^2} \\
\frac{\partial^2 \Phi(x_i, y_1)}{\partial y^2} & \approx \frac{\Phi(x_i, y_2) - 2\Phi(x_i, y_1) + 0}{(\Delta y)^2} \\
\frac{\partial^2 \Phi(x_i, y_{N_y})}{\partial y^2} & \approx \frac{0 - 2\Phi(x_i, y_{N_y}) + \Phi(x_i, y_{N_y-1})}{(\Delta y)^2}
\end{align*}
\]

1.3.2 Periodic Boundary Conditions

Assume that function values from outside of the grid can be taken from the opposite side of the grid (meaning there are compute interactions on the boundary).

\[
\begin{align*}
\frac{\partial \Phi(x_1, y_j)}{\partial x} & \approx \frac{\Phi(x_2, y_j) - \Phi(x_{N_x}, y_j)}{2\Delta x} \\
\frac{\partial \Phi(x_{N_x}, y_j)}{\partial x} & \approx \frac{\Phi(x_1, y_j) - \Phi(x_{N_x-1}, y_j)}{2\Delta x} \\
\frac{\partial \Phi(x_i, y_1)}{\partial y} & \approx \frac{\Phi(x_i, y_2) - \Phi(x_i, y_{N_y})}{2\Delta y} \\
\frac{\partial \Phi(x_i, y_{N_y})}{\partial y} & \approx \frac{\Phi(x_i, y_1) - \Phi(x_i, y_{N_y-1})}{2\Delta y}
\end{align*}
\]
\[
\frac{\partial^2 \Phi(x_1, y_j)}{\partial x^2} \approx \frac{\Phi(x_2, y_j) - 2\Phi(x_1, y_j) + \Phi(x_N_x, y_j)}{(\Delta x)^2}
\]
\[
\frac{\partial^2 \Phi(x_{N_x}, y_j)}{\partial x^2} \approx \frac{\Phi(x_1, y_j) - 2\Phi(x_{N_x}, y_j) + \Phi(x_{N_x-1}, y_j)}{(\Delta x)^2}
\]
\[
\frac{\partial^2 \Phi(x_i, y_1)}{\partial y^2} \approx \frac{\Phi(x_i, y_2) - 2\Phi(x_i, y_1) + \Phi(x_i, y_{N_y})}{(\Delta y)^2}
\]
\[
\frac{\partial^2 \Phi(x_i, y_{N_y})}{\partial y^2} \approx \frac{\Phi(x_i, y_{1}) - 2\Phi(x_i, y_{N_y}) + \Phi(x_i, y_{N_y-1})}{(\Delta y)^2}
\]

### 1.3.3 Neumann Boundary Conditions

This condition is used when a function should be continuous at the boundary. That is, the first-order derivative is continuous and the second-order derivative is zero (meaning there is a constant flux entering or leaving on the boundary).

\[
\frac{\partial \Phi(x_1, y_j)}{\partial x} \approx \frac{\Phi(x_2, y_j) - \Phi(x_1, y_j)}{\Delta x}
\]
\[
\frac{\partial \Phi(x_{N_x}, y_j)}{\partial x} \approx \frac{\Phi(x_{N_x}, y_j) - \Phi(x_{N_x-1}, y_j)}{\Delta x}
\]
\[
\frac{\partial \Phi(x_i, y_1)}{\partial y} \approx \frac{\Phi(x_i, y_2) - \Phi(x_i, y_1)}{\Delta y}
\]
\[
\frac{\partial \Phi(x_i, y_{N_y})}{\partial y} \approx \frac{\Phi(x_i, y_{N_y}) - \Phi(x_i, y_{N_y-1})}{\Delta y}
\]

\[
\frac{\partial^2 \Phi(x_1, y_j)}{\partial x^2} \approx 0
\]
\[
\frac{\partial^2 \Phi(x_{N_x}, y_j)}{\partial x^2} \approx 0
\]
\[
\frac{\partial^2 \Phi(x_i, y_1)}{\partial y^2} \approx 0
\]
\[
\frac{\partial^2 \Phi(x_i, y_{N_y})}{\partial y^2} \approx 0
\]
- Derivative matrix with Dirichlet BC

\[
D_x = \frac{1}{2\Delta x} \begin{bmatrix}
1 & -1 & -1 & \cdots & -1 \\
-1 & 1 & -1 & \cdots & -1 \\
-1 & -1 & 1 & \cdots & -1 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
-1 & -1 & -1 & \cdots & 1
\end{bmatrix} \begin{bmatrix}
\Phi_{00} \\
\Phi_{10} \\
\Phi_{01} \\
\Phi_{11} \\
\Phi_{02} \\
\Phi_{12} \\
\vdots \\
\Phi_{03} \\
\Phi_{13} \\
\Phi_{23} \\
\Phi_{33}
\end{bmatrix}
\]

\[
D_y = \frac{1}{2\Delta y} \begin{bmatrix}
1 & -1 & -1 & \cdots & -1 \\
-1 & 1 & -1 & \cdots & -1 \\
-1 & -1 & 1 & \cdots & -1 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
-1 & -1 & -1 & \cdots & 1
\end{bmatrix} \begin{bmatrix}
\Phi_{00} \\
\Phi_{10} \\
\Phi_{01} \\
\Phi_{11} \\
\Phi_{02} \\
\Phi_{12} \\
\vdots \\
\Phi_{03} \\
\Phi_{13} \\
\Phi_{23} \\
\Phi_{33}
\end{bmatrix}
\]
- Derivative matrix with Periodic BC

\[
D_x = \frac{1}{2\Delta x} \begin{bmatrix}
1 & -1 & & & & \\
-1 & 1 & & & & \\
& -1 & 1 & & & \\
& & -1 & 1 & & \\
& & & -1 & 1 & \\
& & & & -1 & 1 \\
& & & & -1 & 1 \\
1 & -1 & & & &
\end{bmatrix}
\]

\[
D_y = \frac{1}{2\Delta y} \begin{bmatrix}
1 & 1 & 1 & & & \\
& -1 & 1 & & & \\
-1 & 1 & 1 & & & \\
-1 & & -1 & 1 & & \\
-1 & & & -1 & 1 & \\
-1 & & & & -1 & 1 \\
1 & & & & & -1 \\
1 & & & & & -1 \\
1 & & & & & -1
\end{bmatrix}
\]
- Derivative matrix with Neumann BC

\[
D_x = \frac{1}{2\Delta x} \begin{bmatrix}
-2 & 2 & 0 & 0 \\
-1 & 0 & 1 & 0 \\
0 & -1 & 0 & 1 \\
0 & 0 & -2 & 2 \\
\end{bmatrix}
= \begin{bmatrix}
\Phi_{00} \\
\Phi_{10} \\
\Phi_{20} \\
\Phi_{30} \\
\end{bmatrix}
\]

\[
D_y = \frac{1}{2\Delta y} \begin{bmatrix}
-2 & 0 & 0 & 0 & 2 & 0 & 0 \\
0 & -2 & 0 & 0 & 0 & 2 & 0 \\
0 & 0 & -2 & 0 & 0 & 0 & 2 \\
0 & 0 & 0 & -2 & 0 & 0 & 0 \\
\end{bmatrix}
= \begin{bmatrix}
\Phi_{00} \\
\Phi_{01} \\
\Phi_{02} \\
\Phi_{03} \\
\end{bmatrix}
\]
1.4 Linear Algebra

1.4.1 Overdetermined system

A linear system of equations is considered overdetermined if there are more equations than unknowns. Consider a system of linear equations

\[ Ax = b \]

where matrix \( A \) is an \( m \times n \) matrix of rank \( n \), with \( m > n \). Such a system is said to be overdetermined. An overdetermined system is almost always inconsistent (it has no solution) when constructed with random coefficients. However, an overdetermined system will have solutions in some cases, for example if some equation occurs several times in the system or if some equations are linear combinations of the others.

An example in two dimensions: Consider the following system of 3 equations and 2 unknowns (\( x \) and \( y \)), which is overdetermined because \( 3 > 2 \):

\[
\begin{align*}
2x + y &= -1 \\
-3x + y &= -2 \\
-x + y &= 1 
\end{align*}
\]

Any system of linear equations can be written as a matrix equation. For example the previous system of equations can be written as follows:

\[
\begin{bmatrix}
2 & 1 \\
-3 & 1 \\
-1 & 1 
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix}
= 
\begin{bmatrix}
-1 \\
-2 \\
1 
\end{bmatrix}
\]

1.4.2 Least Squares

An overdetermined system of linear equations \( Ax = b \) has more equations than unknowns. The usual reason is too many equations, some equation occurs several times or
some equations are linear combinations of the others. The matrix $A$ has more rows ($m$) than columns ($n$). There are more equations than unknowns, i.e., $m > n$.

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad i = 1, 2 \cdots, m, \quad m \geq n \geq 1$$

This can be written in matrix form, where

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Such a system usually has no exact solution, so the goal is to find the coefficients $x$ that best fit the equations, in the sense of solving the quadratic minimization of residuals $r$ problem.

$$\hat{x} = \text{arg min}_x S(x)$$

where the objective function $S$ is given by

$$S(x) = \sum_{i=1}^{m} r^2 = \sum_{i=1}^{m} \left| b_i - \sum_{j=1}^{n} a_{ij} x_j \right|^2 = \|b - Ax\|^2.$$ 

The matrix $A = a_{ij}$ must have rank $n$, that is, its columns must be linearly independent. We cannot always reduce the error $e = b - Ax$ down to zero. When $e$ is zero, $x$ is an exact solution to $Ax = b$ and when $e$ is as small as possible, $\hat{x}$ is a least squares solution. This minimization problem has a unique solution, provided that the $n$ columns of matrix $A$ are linearly independent, and is given by solving the normal equation.

$$(A^T A) \hat{x} = (A^T b)$$

### 1.5 Electromagnetic Fields

Spatially variant devices interact with electromagnetic waves throughout the inhomogeneities contained on the device to produce a response in the material that affects the
electromagnetic wave. Since electromagnetic waves cannot be controlled inside a homogeneous material, metamaterial devices that are macroscopically homogeneous are very limited in control of electromagnetics waves. The interaction of the electromagnetic waves with the spatially variant device is represented throughout by the permeability $\mu$, which measures the ability of a material to store a magnetic energy and the permittivity $\varepsilon$, which measures how the electric field affects, and is affected by, a dielectric medium. Each of these properties is represented as a function of position and used to spatially vary different attributes of the lattice, such as the structure within the unit cell, fill fraction, lattice spacing, orientation of the unit cells, material composition, and even the symmetry of the lattice [17, 20].

1.6 Spatially Varying a Planar Grating

A transversal electromagnetic oscillation travels at the speed $c \approx 3.00 \times 10^8$ m/s and propagates in the direction of the wave vector $\vec{k}$. A monochromatic (single frequency) wave can be mathematically represented by

$$\vec{\psi}(\vec{r}, t) = \vec{\psi}_o(\vec{r}, t) \cos \left( \vec{k} \cdot \vec{r} - \omega t - \phi \right)$$

where $\vec{\psi}_o$ is the wave amplitude and polarization, $\vec{k}$ the wave vector, $\vec{r}$ the position vector, $\omega$ the angular frequency, and $\phi$ the starting phase angle in radians. Consider the complex wave function

$$\vec{\psi}(\vec{r}, t) = \vec{\psi}_o(\vec{r}, t)e^{i(\vec{k} \cdot \vec{r} - \omega t - \phi)}$$

As the wave propagates, it accumulates phase, which is a function that increases in the direction of wave vector $\vec{k}$.

$$\phi(\vec{r}) = \vec{k} \cdot \vec{r} - \phi$$

Letting the propagating wave be a function of the position and independent of time with a starting phase $\phi = 0$ and phase function $\phi(\vec{r}) = \vec{k} \cdot \vec{r}$:

$$\vec{\psi}(\vec{r}) = \vec{\psi}_o(\vec{r}) \cos [\phi(\vec{r})]$$
Making an analogy between the permittivity function $\varepsilon(\vec{r})$ and the wave propagating function $\vec{\psi}(\vec{r})$, the grating vector $\vec{K}$ serves a similar purpose as the wave vector $\vec{k}$ does for the wave function. Then we can make the following analogy:

$$\vec{k}(\vec{r}) \leftrightarrow \vec{K}(\vec{r})$$

$$\phi(\vec{r}) \leftrightarrow \Phi(\vec{r})$$

$$\vec{\psi}(\vec{r}) \leftrightarrow \varepsilon(\vec{r})$$

Next, we use the permittivity $\varepsilon(\vec{r})$ as a function of the position $\vec{r}$ to define a planar grating in the $\vec{K}$ direction

$$\varepsilon(\vec{r}) = \varepsilon_{\text{avg}} + \Delta \varepsilon \cos \left( \vec{K} \cdot \vec{r} \right)$$

where $\varepsilon_{\text{avg}}$ is the average permittivity, $\Delta \varepsilon$ is the dielectric contrast (amplitude), $\vec{K}$ is the grating vector, and $\vec{r}$ is the position vector. Let us introduce the grating phase $\Phi(\vec{r})$ as an intermediate parameter [19], i.e.,

$$\Phi(\vec{r}) = \vec{K} \cdot \vec{r} = K_x x + K_y y + K_z z \quad (1.7)$$

calculating the gradient

$$\nabla \Phi(\vec{r}) = \vec{K}(\vec{r}) \quad (1.8)$$

As shown by Equation 1.7, the grating phase $\Phi(\vec{r})$ is a scalar quantity that is related to the grating vector $\vec{K}$ through the gradient operation.

As shown by Equation 1.8, the direction of $\vec{K}(\vec{r})$ is to be varied as a function of position $\vec{r}$. The grating vector function is calculated from the desired orientation of the unit cell as a function of position $\theta(\vec{r})$ and the desired period as a function of position $\Lambda(\vec{r})$, where $\theta(\vec{r})$ and $\Lambda(\vec{r})$ are 3D arrays [17, 20].

$$\vec{K}(\vec{r}) = \frac{2\pi}{\Lambda(\vec{r})} \{ \hat{x} \cos[\theta(\vec{r})] + \hat{y} \sin[\theta(\vec{r})] \}$$
1.6.1 Solving the Gradient Equation

Solving the gradient equation can be tricky because all three components of $\vec{K}(\vec{r})$ are trying to be controlled independently with just the grating phase $\Phi(\vec{r})$. Expanding the grating phase gradient equation, we get:

$$\nabla \Phi(\vec{r}) = \vec{K}(\vec{r})$$

$$\frac{\partial \Phi(\vec{r})}{\partial x} \hat{x} + \frac{\partial \Phi(\vec{r})}{\partial y} \hat{y} + \frac{\partial \Phi(\vec{r})}{\partial z} \hat{z} = K_x \hat{x} + K_y \hat{y} + K_z \hat{z}$$

Rewriting the gradient equation in a way that we can easily recognize where the derivative operator can be introduced, we get

$$\begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \Phi(\vec{r}) = \begin{bmatrix} K_x \\ K_y \\ K_z \end{bmatrix}$$

Below we express the previous equation in matrix form using the finite-difference method. The terms $K_x, K_y$, and $K_z$ are column vectors containing the components of the spatially variant grating vectors at each point on the grid throughout the lattice. $D_x, D_y, \text{ and } D_z$ are sparse banded matrices that calculate derivatives across the grid.

$$\begin{bmatrix} D_x \\ D_y \\ D_z \end{bmatrix} \Phi = \begin{bmatrix} K_x \\ K_y \\ K_z \end{bmatrix}$$

This equation can be solved numerically by the least-squares method since the system is overdetermined. Multiplying both sides by $G^T$, we get:

$$G \Phi = K$$

$$(G^T G) \Phi = (G^T K)$$

Considering the solution for the grating phase $\Phi(\vec{r})$, the permittivity is calculated using

$$\varepsilon(\vec{r}) = \varepsilon_{avg} + \Delta \varepsilon \cos[\Phi(\vec{r})]$$
1.7 Why the Grating phase $\Phi(\vec{r}) = \vec{K} \cdot \vec{r}$ fails

Consider the permittivity equation (1.6), i.e.,

$$\varepsilon(\vec{r}) = a \cos [\Phi(\vec{r})]$$  \hfill (1.9)

The term $\cos[\Phi(\vec{r})]$ in Equation (1.9) fails to correctly reconstruct the spatially variant grating when the grating vector is a function of position $\vec{r}$. Suppose we wish to generate the following chirped grating, which is shown in Figure 1.6 [18].

$$\Lambda(z) = \Lambda_0 z \quad 1 \leq z \leq 2$$

$$K(z) = \frac{2\pi}{\Lambda_0 z} = \frac{K_0}{z}$$

where

$$K_0 = \frac{2\pi}{\Lambda_0}$$

$$\Phi(z) = K(z) \cdot z = \frac{K_0}{z} \cdot z = K_0$$

$$\cos[\Phi(z)] = \cos[K_0] = \text{constant}$$

Figure 1.6: Chirped grating as a $\Lambda(z)$

Figure 1.7: Constant values of the Chirped grating function

The argument cosine (grating phase $\Phi(z)$) is a constant and fails to form any kind of chirped grating. As shown in Figure 1.7, consider the grating vector through the intermediate step (gradient operation)

$$\nabla \Phi(\vec{r}) = \vec{K}(\vec{r})$$
Solving for one component of the gradient

\[
\frac{d\Phi}{dz} = \frac{K_0}{z}
\]

\[
\int_{-\infty}^{z} d\Phi = \int_{-\infty}^{z} \frac{K_0}{z} dz
\]

\[
\Phi(z) - \Phi(-\infty) = \int_{-\infty}^{1} \frac{K_0}{z} dz + \int_{1}^{z} \frac{K_0}{z} dz
\]

then

\[
\Phi(z) - \Phi(-\infty) = K_0 \log z|_{-\infty}^{0} + K_0 \log z|_{1}^{z}
\]

In the last equation above, the first term on the right-hand side is a constant. Since it is a phase, we are free to choose whatever is convenient. In this case, we choose zero.

\[
\Phi(z) = K_0 \log z|_{1}^{z} = K_0 [\log z - 0]
\]

Then the permittivity function can be written as Equation 1.10, and is shown in Figure 1.8

\[
\epsilon(z) = a \cos (K_0 \log z) \quad (1.10)
\]

Figure 1.8: Chirped grating permittivity function
1.8 Spatially Varianing Grating Examples

Figure 1.9: Direct reconstruction of uniforme K-Field over an area of $10a \times 10a$ $A = 25^\circ$

Figure 1.10: Direct construction of spatial variant K-Field over an area of $10a \times 10a$
Figure 1.11: Reconstruction using grating phase

Figure 1.12: Reconstruction using grating phase
Chapter 2

3D Spatially Variant Lattice Program
Code Implementation

2.1 Motivation

Here we explain the implementation of the 3D SVL algorithm. The 3D SVL program code is a computationally intensive C code that requires the use of parallel tools (PETSc, FFTW, and MPI), and a parallel computer system such as the one we used here, TACC Stampede2. The SVL code has been programmed using the C language because C is one of the most widely used programming languages with many resources available. The SVL 3D version has been written using the 2D version as a base template. This does not mean that the two codes are similar. In fact, the addition of one dimension requires the use of a few tricks and modifications in order to make the code work properly. Before we start it is important to point out few clarifications to avoid misunderstanding and confusion.

- We use the terms array and matrix interchangeably. This can lead us to a misunderstanding or confusion because we refer to one-dimensional (1D) arrays (vectors), two-dimensional (2D) array as an array of 1D arrays (matrices), and three-dimensional (3D) array as an array of 2D arrays of 1D arrays (3D matrices), and simply array when the number of dimensions is unspecified or unimportant.

- Matrix operations follow the rules of linear algebra. Array operations execute element by element and support multidimensional arrays.
• The linear system solved here is of the form \( G\Phi = K \), where \( G \) and \( K \) are 3D matrices. Therefore we need to rearrange \( G \) into 2D matrix and \( K \) into 1D matrix form.

• Since the 3D SVL is a three-dimensional (3D) implementation, we have the options of switching between cylindrical coordinates, spherical coordinates, or a combination of both coordinates system. To avoid any misunderstanding, we use cylindrical coordinates in this project.

2.2 SVL 3D Program Code

2.2.1 Unit Cell Construction

Similar to the planar grating (2D SVL) parallel implementation, we start by designing the baseline 3D unit cell as a function of position. First, we create a simple 3D zero array and next we fill this array with ones anywhere that dielectric is to be placed to construct a solid 3D unit cell inside the 3D zeros array (See Figure 2.1, and Pseudocode 2).

![3D Unit Cell](figure.png)

Figure 2.1: 3D Unit Cell [18]
2.2.2 Fourier Transform

From the beginning, the unit cell used in this project is assumed to be part of a unit cell periodic structure and it can be represented as periodic functions. Hence it can be decomposed into a complex Fourier series. Therefore, we can rewrite the permittivity \( \varepsilon(\vec{r}) \) as a complex Fourier series, the position as \( \vec{r} = (x, y, z) \), and grating vector as \( \vec{K}_{pqr} = (K_x, K_y, K_z) \) [17, 19]:

\[
\varepsilon(\vec{r}) = \sum_p \sum_q \sum_r a_{pqr} \exp\left(j \vec{K}_{pqr} \cdot \vec{r}\right) \tag{2.1}
\]

\[
\vec{K}_{pqr} \cdot \vec{r} = \frac{2\pi p}{\Lambda_x} x + \frac{2\pi q}{\Lambda_y} y + \frac{2\pi r}{\Lambda_z} z
\]

Any periodic structure can be decomposed into a set of superimposed planar gratings, each having a different complex amplitude \( a_{pqr} \), where each term in the Fourier series is a planar grating (defined by subscript index \( pqr \)) and has its own unique period, complex amplitude, and direction [17, 19].

- \( a_{pqr} = FFT[\varepsilon(\vec{r})] \) is the complex amplitude of the \( (pqr)\)th planar grating (See Figure 2.2).

\[
a_{pqr} = \frac{1}{V} \sum_x \sum_y \sum_z \varepsilon(r) \exp\left(-j \vec{K}_{pqr} \cdot \vec{r}\right) \tag{2.2}
\]

The discrete form of the Fourier transform of a 3D array in Cartesian coordinates can be written in the following form, where \((N_x, N_y, N_z)\) is the size of the unit cell array. As we mentioned before, we use the Fastest Fourier Transform in the West (FFTW) library tools [2] to compute the FFT of a 3D array along all its dimensions, rows for the vector on the \( z\)-axis, columns for the vector on the \( y\)-axis, and heights for the vector on the \( x\)-axis (a vector is a 1D array).

\[
r \leftarrow \text{rows}
\]
\[
q \leftarrow \text{columns}
\]
\[
p \leftarrow \text{heights}
\]
Since our code has been written in the C language, the 3D array is stored as a 1D array in a row-wise fashion. For example,

```c
for (i = 0; i < Nx; i++) { // Loop through the height.
    for (j = 0; j < Ny; j++) { // Loop through the rows.
        for (k = 0; k < Nz; k++) { // Loop through the columns.
            3D_array[i * Ny * Nz + j * Nz + k] = i * Ny * Nz + j * Nz + k;
        }
    }
}
```

**Compute FFTW**

The simplest way to compute a 3D FFT is to consider an array of size $N_x \times N_y \times N_z$ along axes $x$, $y$ and $z$. Since the code has been written using C language, the 3D array is stored in memory in row-major order. Thus, cells along the $z$-axis are contiguous in memory and form a vector that is stored contiguously along the $y$-axis. Those line vectors, in turn, form thin slabs that are stored contiguously. Those data slices (called slabs) are 2D arrays stored contiguously along the $x$-axis. This method is called slab
decomposition or 1D decomposition because data is split along one dimension [16].

**For Loop Arrangement**

To explain how we implement the 3D FFT, let us first understand how the loops arrangement of a 3D array is done using multiple for-loops. Our 3D array is an array of size $A[N_x][N_y][N_z]$ and it is stored in 1D array of size $A[N_x \times N_y \times N_z]$. The for-loop gives us ways to manipulate the arrangement of the 3D array.

```c
for (i = 0; i < Nx; i++) { // Loop through the height.
    for (j = 0; j < Ny; j++) { // Loop through the rows.
        for (k = 0; k < Nz; k++) { // Loop through the columns.
            3D_array[i * Ny * Nz + j * Nz + k] = i * Ny * Nz + j * Nz + k;
        }
    }
}
```

As an example, let us assume a 3D array of size $A[3][3][3]$ which is stored as a 1D array of size $A[27]$.

![3D array arrangement](image)

Figure 2.3: 3D array arrangement
Each box in the 3D array stores only one real or complex numerical value. For example, let us consider a 3D array with dimensions \((N_x \times N_y \times N_z) = (3 \times 3 \times 3)\). Therefore we have the columns along the \(z\)-axis (1D arrays of size 1), the rows along the \(y\)-axis (1D arrays of size 3), and the slab along the \(x\)-axis (2D arrays of size 3 \times 3).

### Table 2.1: Slab \((i = 0)\)

<table>
<thead>
<tr>
<th>(i)</th>
<th>(j)</th>
<th>(k)</th>
<th>(i \times N_y \times N_z + j \times N_z + k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 2.2: Slab \((i = 1)\)

<table>
<thead>
<tr>
<th>(i)</th>
<th>(j)</th>
<th>(k)</th>
<th>(i \times N_y \times N_z + j \times N_z + k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

### Table 2.3: Slab \((i = 2)\)

<table>
<thead>
<tr>
<th>(i)</th>
<th>(j)</th>
<th>(k)</th>
<th>(i \times N_y \times N_z + j \times N_z + k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

**Figure 2.4: From 3D array to 1D array**

**Method**

1. Store the 3D array of size \(U[N_x][N_y][N_z]\) in a 1D array

\[
\text{PetscScalar } U[N_x\times N_y \times N_z];
\]

2. Compute 3D FFT of the array \(U[N_x \times N_y \times N_z]\)

\[
\text{ierr = SVL_3D_FFTW(Nx, Ny, Nz, U);}\text{CHKERRQ(ierr);};
\]
2.2.3 Swap Quadrants

After the 3D Fourier transform is done, the next step is to swap the block quadrants on the 3D array, similar to what already did in the 2D swap (See Figure 2.5) in which we divide the rectangular array in four even parts and swap across the diagonals. For the 3D output array, we have eight quadrants of data that need to be sorted in the following way: back to front, left to right, and up to down (See figure 2.6).

Figure 2.5: 2D Swap

Figure 2.6: 3D Swap

Figure 2.6 shows how the 3D swap process is done. First, we split the 3D array in half along the x-axis and swap the block or move back to front; second we repeat the process, split the 3D array in half along y-axis and move the block left to right, and split the 3D array in half along the z-axis and move the block up to down. This process results in moving the dots initialized at the array corners to the center of the 3D array.

2.2.4 Truncate Spatial Harmonics

The truncation process consists of reducing the original spatial harmonics array to a smaller array that contains just the most significant harmonics of the FFT. After the swap is done, the highest harmonics values are at the center of the 3D array. The size of the truncated array needs to be enough to make possible the reconstruction of the original unit cell with a minor loss in geometry. The quality of the unit cell reconstruction improves as we add more harmonic components. We chose a 3D array size of $(NM \times NN \times NP) = (13 \times 13 \times 13)$ spatial harmonics which is big enough to get a decent reconstruction of the unit cell device. The following figure shows the 3D truncation
2.2.5 Grating Vector

Each planar grating in equation 2.1 is described by the grating vector $K_{pqr}$ and a complex amplitude $a_{pqr}$. The grating vector $K_{pqr}$ associated with the spatial harmonics is calculated using equation 2.3 [19].

$$\vec{K}_{pqr} = K_x \hat{x} + K_y \hat{y} + K_z \hat{z} = \frac{2\pi}{\Lambda_x} \hat{x} + \frac{2\pi}{\Lambda_y} \hat{y} + \frac{2\pi}{\Lambda_z} \hat{z}$$ (2.3)

Equation 2.3 shows the grating components in Euclidean space using Cartesian coordinates. Each component of the grating vector can be separated and represented by a 3D array of multiple 2D arrays of the same shape that have been piled together to build the 3D array component. The following figures show cross sections of the 3D array.
2.2.6 Device Parameters

The lattice parameters \((N_{P_x}, N_{P_y}, N_{P_z})\) are the number of unit cells used to build a 3D lattice structure, for example:

\[
(N_{P_x}, N_{P_y}, N_{P_z}) = (7, 7, 7)
\]

and the parameters \((N_{GP_x}, N_{GP_y}, N_{GP_z})\) are the number used to build each unit cell on the 3D lattice, for example:

\[
(N_{GP_x}, N_{GP_y}, N_{GP_z}) = (5, 5, 5)
\]

The device dimension is determined by the new values of \((\tilde{N}_x, \tilde{N}_y, \tilde{N}_z)\), which are the number of points on the grid.

\[
\tilde{N}_x = N_{GP_x} N_{P_x}
\]
\[
\tilde{N}_y = N_{GP_y} N_{P_y}
\]
\[
\tilde{N}_z = N_{GP_z} N_{P_z}
\]
\[ |\vec{K}| = \frac{2\pi}{\Lambda} \]

\[ \vec{K} = K_x \hat{x} + K_y \hat{y} + K_z \hat{z} \]

Figure 2.9: Grating Vector \( \vec{K} = (K_x, K_y, K_z) \) [18]

\[ K_x(\cdot,0,\cdot) = \frac{2\pi}{\Lambda_x} \begin{bmatrix} -6 & -5 & -4 & -3 & -2 & -2 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ -6 & -5 & -4 & -3 & -2 & -2 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -6 & -5 & -4 & -3 & -2 & -2 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ -6 & -5 & -4 & -3 & -2 & -2 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \]

Figure 2.10: \( K_x \) Grating Vector component and cross section xz-plane

\[ K_y(0,\cdot,\cdot) = \frac{2\pi}{\Lambda_y} \begin{bmatrix} -6 & -5 & -4 & -3 & -2 & -2 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ -6 & -5 & -4 & -3 & -2 & -2 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -6 & -5 & -4 & -3 & -2 & -2 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ -6 & -5 & -4 & -3 & -2 & -2 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \]

Figure 2.11: \( K_y \) Grating Vector component and cross section yz-plane
The grid size is \((S_x, S_y, S_z) = (1, 1, 1)\) and the device resolution is the following:

\[
\begin{align*}
\tilde{d}_x &= \frac{N_{P_x} S_x}{\tilde{N}_x} \\
\tilde{d}_y &= \frac{N_{P_y} S_y}{\tilde{N}_y} \\
\tilde{d}_z &= \frac{N_{P_z} S_z}{\tilde{N}_z}
\end{align*}
\]

For the parameters shown above, the 3D lattice contains a total of \(7^3 = 343\) unit cells, each unit cell is built with \(5^3 = 125\) points, and the lattice dimension is \((\tilde{N}_x, \tilde{N}_y, \tilde{N}_z) = (35 \times 35 \times 35)\).

As explained in Chapter 4, the parameters \(N_{P}\) and \(N_{GP}\) are used to increase the number of points in the lattice and change the density of the lattice since we keep the grid size constant \((S = 1)\). The parameter \(N_{P}\) is used to change the number of unit cells in the lattice and \(N_{GP}\) to change the number of points used to build each unit cell.

### 2.2.7 Spatially Varying the Lattice

Each spatial attribute (unit cell orientation, lattice spacing, and fill fraction) is constructed as a separate function of position that varies throughout the 3D volume of the
lattice. This implementation is done by using cylindric \((r, \theta, z)\) or spherical coordinates \((r, \theta, \varphi)\). The cell orientation \(\Theta\) and lattice spacing \(\text{PER}\) attribute are defined prior to the synthesis procedure and spatially varied in the iterative loop at the same time. The fill fraction is implemented at the end over the final lattice product [19].

### 2.2.8 Orientation Attribute

The orientation attribute is a 3D array function. Below we show cross sections of the 3D arrays for \(X, Y\) and \(Z\).

Using Cartesian coordinates:

\[
\text{RSQ} = (X - \bar{X})^2 + (Y - \bar{Y})^2
\]

\[
\text{THETA} = \text{atan2}(Y, X)
\]
In cylindrical coordinates, RSQ squares the component positions, and the axes rotation in two dimensions is a mapping from \(xy\) Cartesian coordinates to \(x'y'\) cylindrical coordinate in which the origin is kept fixed and the \(x'\)-axis and \(y'\)-axis are obtained by rotating the \(x\)-axis and \(y\)-axis counterclockwise through an angle \(\Theta\).

![Cylindrical coordinates](image)

**Figure 2.16: Cylindrical coordinates**
Figure 2.17: Cylindrical coordinates : RSQ array

Figure 2.18: Cylindrical coordinates : THETA array
2.2.9 Lattice Spacing

We can compute every element of the lattice spacing array by using the following equations:

\[
\text{RSQ}_{ijk} = (X_{ijk} - \bar{X})^2 + (Y_{ijk} - \bar{Y})^2
\]

where \(\bar{X}\) and \(\bar{Y}\) are the mean values in each component direction. The lattice spacing attribute \(\text{PER}\) is an 3D array function with the same dimensions as the \(\Theta\) array that has the form of a Gaussian distribution array over each cross section \((X, Y)\).

\[
(a_{ij})_k = 1 + \exp\left(-\frac{\text{RSQ}_{ijk}}{(2\alpha)^2}\right)
\]

\[
\text{PER} = \begin{bmatrix}(a_{ij})_k\end{bmatrix}
\]

![PER array](image)

Figure 2.19: Cylindrical coordinates: PER array

2.2.10 Finite-Difference Derivative Operators (FDDER) for 3D

The SVL technique consists of solving the gradient equation

\[
\nabla \Phi(\vec{r}) = \vec{K}(\vec{r})
\]
The gradient is built as a first-order derivative and various boundary conditions such as: Dirichlet, Neumann, and Periodic.

\[ \nabla = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} = \begin{bmatrix} D_x \\ D_y \\ D_z \end{bmatrix} \]

Let us show the derivative for \((3 \times 3 \times 3)\) with Neumann BC and \(\Delta x = \Delta y = \Delta z = 1\).
\[
\begin{bmatrix}
-2 & 2 & 0 \\
-1 & 0 & 1 \\
0 & -2 & 2
\end{bmatrix}
\]

\[
D_x = \frac{1}{2\Delta x}
\]

\[
\begin{bmatrix}
\phi(0,0) \\
\phi(1,0) \\
\phi(2,0) \\
\phi(0,1) \\
\phi(1,1) \\
\phi(2,1) \\
\phi(0,2) \\
\phi(1,2) \\
\phi(2,2)
\end{bmatrix}
\]
\[
D_y - \frac{1}{2\Delta y} \begin{bmatrix}
-2 & 0 & 0 & \ldots & 2 & 0 & 0 \\
0 & -2 & 0 & \ldots & 0 & 2 & 0 \\
0 & 0 & -2 & \ldots & 0 & 0 & 2 \\
-1 & 0 & 0 & \ldots & 1 & 0 & 0 \\
0 & -1 & 0 & \ldots & 0 & 1 & 0 \\
0 & 0 & -1 & \ldots & 0 & 0 & 1 \\
-2 & 0 & 0 & \ldots & 2 & 0 & 0 \\
0 & -2 & 0 & \ldots & 0 & 2 & 0 \\
0 & 0 & -2 & \ldots & 0 & 0 & 2 \\
\end{bmatrix} 
\]
$$D_x = \frac{1}{\Delta x}$$

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>-2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>-2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>-2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>-2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>-2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>-2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>-2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Each derivative matrix is a sparse matrix of size $M \times M$, where is $(M = \tilde{N}_x \cdot \tilde{N}_y \cdot \tilde{N}_z)$. The main derivative operator $G$ is built as a rectangular sparse array using the gradient elements $(D_x, D_y, D_z)$, and the size of the $G$ operator array is $(3M \times M)$ (See Figure 2.20 and Table 2.6).

$$G = \begin{bmatrix}
\frac{\partial}{\partial x} \\
\frac{\partial}{\partial y} \\
\frac{\partial}{\partial z}
\end{bmatrix} = \begin{bmatrix}
D_x \\
D_y \\
D_z
\end{bmatrix}$$

These are used later for solving the overdetermined linear system for the following form

$$G \Phi = K$$

### 2.2.11 SVL Loop

The truncated array $AMNP$ with a predetermined size of $(13 \times 13 \times 13)$ stores the spatial harmonic used in the loop, with a length size of $13^3 = 2197$. Every element of the truncated array $AMNP$ has a corresponding element in the grating matrices $(K_x, K_y, K_z)$. The truncated array $AMNP$ and the grating vector component array $(K_x, K_y, K_z)$ have the same size. The complete cycle of the for-loop will take 2197 loops, assuming we did not eliminate any element on the truncate array and its corresponding grating array. We use the letter $m$ to identify each spatial harmonic.

$$m = 0, \ldots, 2196$$

After the implementation of the elimination of the gratings according to their amplitude, the spatial harmonics in the truncated array that passes the threshold and its corresponded grating vector are reduced to $735 < \text{length}(AMNP)$. These values are stored back and ready to be used to complete the full cycle of the for-loop.

$$m = 0, \ldots, 734$$
Figure 2.20: Gradient operator $G$ array with Newmann BC and $\Delta x = \Delta y = \Delta z = 1$
As we mentioned at the beginning of this chapter 2.1, we do not have linear algebra tools for solving 3D matrix operations. Therefore every 3D matrix inside the for-loop section is rearranged into 2D or 1D matrix form. There are not 3D matrices or arrays anymore inside the for-loop, but the final outputs are saved in 3D array form and later processed and present as 3D figures.

**Table 2.4: Loop matrix dimensions**

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K^m_x$</td>
<td>$M \times 1$</td>
</tr>
<tr>
<td>$K^m_y$</td>
<td>$M \times 1$</td>
</tr>
<tr>
<td>$K^m_z$</td>
<td>$M \times 1$</td>
</tr>
<tr>
<td>$\rho^m$</td>
<td>$M \times 1$</td>
</tr>
<tr>
<td>$\theta^m$</td>
<td>$M \times 1$</td>
</tr>
<tr>
<td>PER</td>
<td>$M \times 1$</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>$M \times 1$</td>
</tr>
<tr>
<td>$D_x$</td>
<td>$M \times M$</td>
</tr>
<tr>
<td>$D_y$</td>
<td>$M \times M$</td>
</tr>
<tr>
<td>$D_z$</td>
<td>$M \times M$</td>
</tr>
<tr>
<td>$G$</td>
<td>$3M \times M$</td>
</tr>
<tr>
<td>$\Phi^m$</td>
<td>$M \times 1$</td>
</tr>
<tr>
<td>$K^m$</td>
<td>$3M \times 1$</td>
</tr>
</tbody>
</table>

a. Construction of the spatially variant lattice starts by building the grating vector associated with the spatial harmonic $K^m = (K_x[m], K_y[m], K_z[m])$, where each component is a 3D array of dimension ($\tilde{N}_x \times \tilde{N}_y \times \tilde{N}_z$) that is rearranged to 1D form with dimension ($M \times 1$) (See Table 2.4).
\[ K^m_x(x, y, z) = K_x[m] \]

\[ K^m_y(x, y, z) = K_y[m] \]

\[ K^m_z(x, y, z) = K_z[m] \]

- Change coordinates:
  - Cartesian \((K^m_x, K^m_y, K^m_z)\) to cylindrical \((\rho^m, \theta^m, K^m_x)\)

\[ \rho^m(x, y, z) = \sqrt{(K^m_y(x, y, z))^2 + (K^m_z(x, y, z))^2} \]

\[ \theta^m(x, y, z) = \tan^{-1} \left( \frac{K^m_z(x, y, z)}{K^m_y(x, y, z)} \right) \]

\[ K^m_x(x, y, z) = K^m_x(x, y, z) \]

- Including the orientation attribute: We change the orientation of the SVL by adding the orientation array \(\Theta\):

\[ \theta^m(x, y, z) = \theta^m(x, y, z) + \Theta \]
• Including the lattice spacing attribute: The lattice spacing array $\text{PER} = [a(x, y, z)]$ stores the Gaussian distribution values used to smooth the SVL, i.e.,

$$\rho^m(x, y, z) = \frac{\rho^m(x, y, z)}{\text{PER}}$$

• Change coordinates:
  - Cylindrical $(\rho^m, \theta^m, K^m_z)$ to cartesian $(K^m_x, K^m_y, K^m_z)$

\[
K^m_x(x, y, z) = K^m_x(x, y, z) \\
K^m_y(x, y, z) = 2\pi \rho^m(x, y, z) \cos[\theta^m(x, y, z)] \\
K^m_z(x, y, z) = 2\pi \rho^m(x, y, z) \sin[\theta^m(x, y, z)]
\]

b. Calculate the grating phase function $\Phi^m$ from the spatially variant $K^m$.

\[
\begin{bmatrix}
D_x \\
D_y \\
D_z
\end{bmatrix}_{3M \times M} \Phi^m = 
\begin{bmatrix}
K_x^m \\
K_y^m \\
K_z^m
\end{bmatrix}_{3M \times 1}
\]

System is overdetermined

\[
G \Phi^m = K^m \\
(G^T G)_{M \times M} \Phi^m = (G^T K^m)_{M \times 1}
\]

c. Add each planar grating to the overall lattice. Generate the spatially variant grating for the $m$ spatial harmonic:

\[
\epsilon^m(x, y, z) = AMNP[m] \exp(j\Phi^m(x, y, z))
\]

Add this to the analog grating. How this process works is shown in Figure 2.21 for the 2D case. The same process is applied for the 3D case.

\[
\epsilon_{\text{analog}}(x, y, z) = \epsilon_{\text{analog}}(x, y, z) + \Re(\epsilon^m(x, y, z)) \quad (2.4)
\]
2.2.12 Form Binary Grating Using Spatially Variant Threshold

The binary grating $\varepsilon_{m}(x, y, z)$ is calculated from the analogue grating equation 2.4 using a threshold function $\gamma(x, y, z)$ to get the desired fill fraction $f(x, y, z)$ function.

$$\varepsilon(x, y, z) = \begin{cases} \varepsilon_1, & \varepsilon_{m}(x, y, z) \leq \gamma(x, y, z) \\ \varepsilon_2, & \varepsilon_{m}(x, y, z) > \gamma(x, y, z) \end{cases}$$

For near sinusoidal lattices, this can be estimated as

$$\gamma(x, y, z) \approx \cos(\pi f(x, y, z))$$

The binary grating is then

$$\varepsilon_{\text{binary}}(x, y, z) = \varepsilon_{\text{analog}}(x, y, z) > \gamma(x, y, z)$$

The fill fraction can be spatially varied simply by making $\gamma(x, y, z)$ a function of position.

$$\gamma(x, y, z) = \cos(\pi f(x, y, z)) \quad \text{(For a sinusoidal lattices)}$$

$$\varepsilon_{\text{binary}}(x, y, z) = \varepsilon_{\text{analog}}(x, y, z) > \gamma(x, y, z) \quad \text{(Binary grating)}$$
2.3 SVL Pseudocode

The SVL program code is divided into subprograms that can be used as one step or as part of another subprogram. This allows us to build the SVL algorithm in sections that can be easy to manipulate. The SVL code is read, compiled and executed by a bash file with extension `.sh`, called script. The Bash is a command processor where the user types a set of commands that cause actions. The Bash executes the Makefile. The Makefile handles the entire set of C programs. The Makefile automatically builds executable programs and libraries from the source code by reading the programs and subprograms integrated into the Makefile. The integration of the C code is possible by a common header file with extension `.h` that contains C subprogram function declarations and macro definitions to be shared among the main program and subprograms. The combination of these computational tools allows the SVL subprograms to work together as they are in a pipeline. Below we outline the pseudocode implementation of the SVL program C/PETSc code. Tables 2.5 and 2.6 show the names of the variables and arrays used in the pseudocode and 3D SVL code.

Table 2.5 describes the array or matrix used throughout the program. Each array was built using for-loops. Even though each array is 3D array inside the program, we move and operate on it as a 1D array. Working everything as 1D help us to handle the 3D arrays with the different tools we used (FFTW and PETSc solvers).

Table 2.6 starts with the variables to construct the grid with dimension ($S_x = S_y = S_z = 1$). This allows us to plot the figures with a fixed size of 1. Next, we have the unit cell size ($N_x, N_y, N_z$) and resolution ($dx, dy, dz$). We keep the shrink variable $a$ constant. The Fourier transform of the unit cell has the same size as the unit cell. The truncated array on the other side is a smaller matrix of size $(NM, NN, NP)$. The full size of the truncated array is recorded in the variable NK. This variable is updated throughout the program. The C operator (&, *) allowed us to update and pass the value of the variables or arrays throughout the subprograms and the main program. Next, we have the lattice
size \((\tilde{N}_x, \tilde{N}_y, \tilde{N}_z)\) and device resolution. The product of the lattice dimension is \(M\) which stores the value of the problem size. Remember that \(M\) is modified by the number of unit cells on the lattice \((NP_x, NP_y, NP_z)\), and the resolution \((NGP_x, NGP_y, NGP_z)\), which is the number of points used to construct each unit cell in the lattice. The C operator “\(*\)” means that each array is addressed throughout the code as a reference and “&” means that the variable is updated and passed as an argument through the subprograms and the main program (See section 2.2).

<table>
<thead>
<tr>
<th>Arrays</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ast U)</td>
<td>Unit cell array</td>
</tr>
<tr>
<td>(\ast AC)</td>
<td>FFTW((U))</td>
</tr>
<tr>
<td>(\ast AMNP)</td>
<td>Truncate FFTW array</td>
</tr>
<tr>
<td>(\ast K_x)</td>
<td>Grating arrays</td>
</tr>
<tr>
<td>(\ast K_y)</td>
<td></td>
</tr>
<tr>
<td>(\ast K_z)</td>
<td></td>
</tr>
<tr>
<td>(\ast RSQ)</td>
<td>Square</td>
</tr>
<tr>
<td>(\ast \text{THETA})</td>
<td>Rotation</td>
</tr>
<tr>
<td>(\ast \text{PER})</td>
<td>Lattice spacing</td>
</tr>
<tr>
<td>(\ast G)</td>
<td>Derivative operator</td>
</tr>
<tr>
<td>(\Phi)</td>
<td>Grating phase</td>
</tr>
<tr>
<td>(S)</td>
<td>(\exp(i\Phi))</td>
</tr>
<tr>
<td>(U_c)</td>
<td>(U_c + S)</td>
</tr>
<tr>
<td>(\ast)</td>
<td>Passed as a reference using operator (*)</td>
</tr>
</tbody>
</table>
Table 2.6: Variables description

<table>
<thead>
<tr>
<th>Variable</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_x = 1$</td>
<td>Grid size</td>
</tr>
<tr>
<td>$S_y = 1$</td>
<td></td>
</tr>
<tr>
<td>$S_z = 1$</td>
<td></td>
</tr>
<tr>
<td>$a = 1$</td>
<td>Shrink or contract parameter</td>
</tr>
<tr>
<td>$N_x$</td>
<td>Unit cell dimensions</td>
</tr>
<tr>
<td>$N_y$</td>
<td></td>
</tr>
<tr>
<td>$N_z$</td>
<td></td>
</tr>
<tr>
<td>$dx$</td>
<td>Unit cell resolution</td>
</tr>
<tr>
<td>$dy$</td>
<td></td>
</tr>
<tr>
<td>$dz$</td>
<td></td>
</tr>
<tr>
<td>NM=13</td>
<td>Truncate array dimension</td>
</tr>
<tr>
<td>NN=13</td>
<td></td>
</tr>
<tr>
<td>NP=13</td>
<td></td>
</tr>
<tr>
<td>&amp;NK</td>
<td>Number of spatial harmonics</td>
</tr>
<tr>
<td>$\tilde{N}_x$</td>
<td></td>
</tr>
<tr>
<td>$\tilde{N}_y$</td>
<td></td>
</tr>
<tr>
<td>$\tilde{N}_z$</td>
<td></td>
</tr>
<tr>
<td>$\tilde{d}x$</td>
<td></td>
</tr>
<tr>
<td>$\tilde{d}y$</td>
<td></td>
</tr>
<tr>
<td>$\tilde{d}z$</td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>$N_x \cdot N_y \cdot N_z$</td>
</tr>
<tr>
<td>&amp;</td>
<td>Passed as an arguments using operator (&amp;)</td>
</tr>
</tbody>
</table>
2.3.1 Algorithm 1: Main Program

The SVL main program is a sequence of instructions that are used to control the SVL algorithm code structure. The main program invokes several subprograms (See Algorithm 1).

The first seven subprograms listed in the main program are serial codes. In the first subprogram, the 3D unit cell array $U$ with size $(N_x, N_y, N_z)$ is constructed. In the second, the Fourier transform of the 3D unit cell is calculated, swapping the quadrant to build that harmonic array $AC$ which contains the Fourier harmonics. In the third, the Fourier transforms array $AC$ is truncated and saved in the array $AMNP$ with size $(NM \times NN \times NP)$.

The variable $NK = NM \cdot NN \cdot NP = 13 \cdot 13 \cdot 13$ records the size of the $AMNP$ array. In the fourth subprogram, the grating arrays $(K_x, K_y, K_z)$ with size $(NM \times NN \times NP)$ are built. In the fifth subprogram, the number of harmonics is reduced by eliminating gratings according to their amplitude improvement. The harmonic values are saved into the array $AMNP$. The size of the array $AMNP$ is reduced to a smaller number $NK << N_x \cdot N_y \cdot N_z$. The new value of $NK$ is updated and passed as an argument, reducing the number of times the LOOP is called. In the sixth subprogram, the attributes arrays square position array $RSQ$, orientation matrix $THETA$, and lattice spacing array $PER$ are built. In the seventh subprogram, the gradient derivative operator $G$ is built with Neumann boundary conditions (See Section 2.2, and Tables 2.6 and 2.5).

In the LOOP, an overdetermined linear system ($m > n$) for the least squares problem is solved using an iterative method (conjugate gradient) $NK$ times. The linear system is solved in a parallelization process by breaking it into small pieces. Finally, the arrays $\Phi$, $S$ and $UC$ are saved (See Section 2.2, and Tables 2.6 and 2.5).
Algorithm 1: Main Program

UNIT CELL($N_x$, $N_y$, $N_z$, *U, $a$, $dx$, $dy$, $dz$)

FFTW SWAP(&NK, $N_x$, $N_y$, $N_z$, *U, *AC)

TRUNCATE ARRAY($N_x$, $N_y$, $N_z$, *AC, *AMNP)

GRATING VECTOR($NM$, $NN$, $NP$, $S_x$, $S_y$, $S_z$, *KK, *KY, *KZ)

IMPLEMENT IMPROVEMENTS($NM$, $NN$, $NP$, &NK, *KK, *KY, *KZ, *AMPN)

CYLINDRIC SPATIAL VARIANT($\tilde{N}_x$, $\tilde{N}_y$, $\tilde{N}_z$, $\tilde{dx}$, $\tilde{dy}$, $\tilde{dz}$, *X, *Y, *Z, *RSQ, *THETA, *PER)

FDDER(NS, BC, RES, *G)


STORE $\Phi$, S, $U_C$

2.3.2 Algorithm 2: Unit Cell

Here we present the 3D unit cell pseudocode. The subprogram is built as follows. First, we build position arrays for each axis. We move the center position to the middle of each array axis. Then these position arrays run from negative to positive (−$m$ to $m$) along each of the axes. Second, we initialize the unit cell array with 0’s. Third, we add the dielectric device to the unit cell array center, using the $\cos(\theta)$ function to shape the structure of the 3D unit cell. Lastly, we fill the unit cell volume by replacing any value higher than 0.8 with 1’s 0.8 and everything else with 0’s. The 0’s represent the empty space and 1’s the dielectric device. Notice that the unit cell is used to build a periodic structure of unit cells. The algorithm below is a serial pseudocode that initializes the 3D unit cell device of 1’s and 0’s.
Algorithm 2: SVL 3D UNIT CELL($N_x$, $N_y$, $N_z$, *U, a, dx, dy, dz*)

1: $\pi \leftarrow 4 \cdot \tan^{-1}(1.0)$
2: $a \leftarrow 1$  \hspace{1cm} \triangleright \text{Shrink or contract}$
3: $K_a \leftarrow \frac{2\pi}{a}$
4: \textbf{procedure} COMPUTE THE POSITION ARRAYS AND SHAPES
5: \hspace{2em} \textbf{for} $i = 0 : N_x - 1$ \textbf{do}
6: \hspace{4em} \textbf{for} $j = 0 : N_y - 1$ \textbf{do}
7: \hspace{6em} \textbf{for} $k = 0 : N_z - 1$ \textbf{do}
8: \hspace{8em} $a_{\text{array}}[iN_yN_z + jN_z + k] \leftarrow (j - N_y/2) \cdot dx$
9: \hspace{8em} $a_{\text{array}}[iN_yN_z + jN_z + k] \leftarrow (k - N_z/2) \cdot dy$
10: \hspace{8em} $a_{\text{array}}[iN_yN_z + jN_z + k] \leftarrow (i - N_x/2) \cdot dz$
11: \hspace{6em} $U[iN_yN_z + jN_z + k] \leftarrow 0$  \hspace{0.5cm} \triangleright \text{Initialize array}$
12: \hspace{4em} \textbf{end for}$
13: \hspace{2em} \textbf{end for}$
14: \hspace{2em} \textbf{end for}$
15: \hspace{2em} \textbf{end procedure}$
2.3.3 Algorithm 3: Implementation of the Fourier Transform

The Fourier transform is implemented using the Fastest Fourier Transform in the West (FFTW) C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions. The FFTW tool allows computing the 3D Fourier Transform of the unit cell. The DFT unit cell calculation is done by invoking the 3D FFTW in 1 step or 1D FFTW over each coordinate axis with multiple steps. We write the FFTW subprogram avoiding the unnecessary use of memory. This is done by setting up memory allocations for the input and output. The input is the unit cell which is a real number array and the Fourier transform output is a complex number array. We can make the input a complex number even though this is not necessarily true. Here we present the FFTW pseudocode using the 3D FFTW and 1D FFTW.
Algorithm 3: 3D FFTW, IFFTW AND SWAP

procedure METHOD 1:
    3D FFTW(Nx, Ny, Nz, *U, *AC)
end procedure

procedure METHOD 2:
    TRANSPOSE Z TO Y(Nx, Ny, Nz, *U)
    for i = 0 : Nx − 1 do
        for j = 0 : Ny − 1 do
            for k = 0 : Nz − 1 do
                Uarray[k] ← U[iNyNz + jNz + k]
            end for
            1D FFTW(Nz, *Uarray, *ACarray)
            for k = 0 : Nz − 1 do
                AC[iNyNz + jNz + k] ← ACarray[k]
            end for
        end for
    end for
    TRANSPOSE Y TO Z(Nx, Ny, Nz, *U)
    for i = 0 : Nx − 1 do
        for j = 0 : Ny − 1 do
            for k = 0 : Nz − 1 do
                Uarray[k] ← AC[iNyNz + jNz + k]
            end for
            1D FFTW(Nz, *Uarray, *ACarray)
            for k = 0 : Nz − 1 do
                AC[iNyNz + jNz + k] ← ACarray[k]
            end for
        end for
    end for
end procedure
Algorithm 3: Continue - 3D FFTW, IFFTW AND SWAP

TRANSPOSE Z TO X($N_x, N_y, N_z, \ast U$)

for $i = 0 : N_x - 1$
do
  for $j = 0 : N_y - 1$
do
    for $k = 0 : N_z - 1$
do
      $U_{\text{array}}[k] \leftarrow AC[iN_yN_z + jN_z + k]$
    end for
  1D FFTW($N_z, *U_{\text{array}}, *AC_{\text{array}}$)
  for $k = 0 : N_z - 1$
do
    $AC[iN_yN_z + jN_z + k] \leftarrow AC_{\text{array}}[k]$
  end for
end for
end for

TRANSPOSE X TO Z($N_x, N_y, N_z, \ast U$)

procedure IFFTW AND SWAP

  3D IFFT($N_x, N_y, N_z, *AC, invU$)
  3D SWAP QUADRANTS($N_x, N_y, N_z, *AC$)

end procedure

The pseudocode in Algorithm 4 describes the 3D FFTW implementation. The code starts by setting the FFTW plan $\text{fftw\_plan}$. Next, we set up memory allocation of size ($N = N_x \cdot N_y \cdot N_z$) for the input and output. We force the action of the $\text{fftw\_plan}$ by executing the following command $\text{fftw\_plan\_dft\_3d}$. Temporary vectors $xa, ya$ and arrays $x_{\text{array}}$ are set up to allow the transfer of the input and output during the FFTW execution. The FFTW is executed by invoking $\text{fftw\_execute}$. Lastly we transfer the output and free our memory allocation and transfer arrays memory. A similar approach is used for the 1D FFTW version with a major change in the allocation of memory $N = N_z$ and $\text{fftw\_plan}$ action by executing the $\text{fftw\_plan\_dft\_1d}$. 

59
Algorithm 4: 3D FFTW($N_x, N_y, N_z, \ast U, \ast AC$)

1: $fftw\ plan \leftarrow f_{\ plan}$ \Comment{Set FFTW Forward plan}
2: $M \leftarrow N_x N_y N_z$
3: $x, y \leftarrow$ vectors
4: $x_{sa}, y_{sa}, x_{array} \leftarrow$ arrays
5: $U_{in} \leftarrow$ complex malloc \Comment{Set Input}
6: $U_{out} \leftarrow$ complex malloc \Comment{Set Output}
7: Create vector sequestial array($N, U_{in}, \& x$) \Comment{Real space vector}
8: Create vector sequestial array($N, U_{out}, \& y$) \Comment{Frequency space vector}
9: procedure SET UP PLAN()
10: 
\begin{align*}
& f_{\ plan} \leftarrow \text{fftw plan dft 3d}(N_x, N_y, N_z, U_{in}, U_{out}, \text{FFTW FORWARD, FFTW ESTIMATE}) \end{align*}
\end{procedure}
11: end procedure
12: procedure INITIALIZE INPUT ARRAY()
13: for $i = 0 : N_x - 1$ do
14: \hspace{1em} for $j = 0 : N_y - 1$ do
15: \hspace{2em} for $k = 0 : N_z - 1$ do
16: \hspace{3em} $x_{array}[iN_yN_z + jN_z + k] \leftarrow U[iN_yN_z + jN_z + k]$
17: \hspace{2em} end for
18: \hspace{1em} end for
19: end for
20: end procedure
21: $x \leftarrow x_{array}$ \Comment{Restore Arrays}
22: $x_{a} \leftarrow x$
23: $y_{a} \leftarrow y$
24: $fftw\ execute \leftarrow (f_{\ plan})$ \Comment{Execute Plan}
25: procedure GET OUTPUT
26: for $i = 0 : N_x - 1$ do
27: \hspace{1em} for $j = 0 : N_y - 1$ do
28: \hspace{2em} for $k = 0 : N_z - 1$ do
29: \hspace{3em} $AC[iN_yN_z + jN_z + k] \leftarrow y_{a}[iN_yN_z + jN_z + k]$
30: \hspace{2em} end for
31: \hspace{1em} end for
32: end for
33: end procedure
34: Free $f_{\ plan}, U_{in}, U_{out}, x, y$ \Comment{Free memory}
35: Return $AC$
Algorithm 5 : 1D FFTW\((N_z, \ast U, \ast AC)\)

1: \texttt{fftw plan} \leftarrow \texttt{f_plan} \quad \triangleright \text{Set FFTW Forward plan}
2: \texttt{N} \leftarrow N_z
3: \texttt{U_{in}} \leftarrow \texttt{complex malloc} \quad \triangleright \text{Set Input}
4: \texttt{U_{out}} \leftarrow \texttt{complex malloc} \quad \triangleright \text{Set Output}
5: \texttt{Create vector sequestial array}(N, \texttt{U_{in}}, \&x) \quad \triangleright \text{Real space vector}
6: \texttt{Create vector sequestial array}(N, \texttt{U_{out}}, \&y) \quad \triangleright \text{Frequency space vector}
7: \texttt{procedure SET UP PLAN()}
8: \texttt{f_plan} \leftarrow \texttt{fftw plan dft 1d}(N_z, \texttt{U_{in}}, \texttt{U_{out}}, \texttt{FFTW FORWARD, FFTW ESTIMATE})
9: \texttt{end procedure}
10: \texttt{procedure INITIALIZE INPUT ARRAY()}
11: \texttt{for} k = 0 : N_z \texttt{- 1 do}
12: \quad \texttt{xarray}[k] \leftarrow U[k]
13: \texttt{end for}
14: \texttt{end procedure}
15: x \leftarrow \texttt{xarray} \quad \triangleright \text{Restore Arrays}
16: x_a \leftarrow x
17: y_a \leftarrow y
18: \texttt{fftw execute} \leftarrow (\texttt{f_plan}) \quad \triangleright \text{Execute Plan}
19: \texttt{procedure GET OUTPUT}
20: \texttt{for} k = 0 : N_z \texttt{- 1 do}
21: \quad \texttt{AC}[k] \leftarrow y_a[k]
22: \texttt{end for}
23: \texttt{end procedure}
24: \texttt{Free} \texttt{f_plan, U_{in}, U_{out}, x, y} \quad \triangleright \text{Free memory}
25: \texttt{Return} \texttt{AC}

2.3.4 Algorithm 5: Loop to Solve the Linear System

The loop subprogram is the most intense computational part of the SVL code. It is where the overdetermined linear system is solved multiple times using the least squares method. Inside this subprogram, the main for-loop iterates over the number of spatial harmonics \(NK\) stored in the truncated array. Hence, the reduction of the spatial harmonics number is important to improve the performance of the SVL code. Therefore, the fewer spatial harmonics considered, the less intense computer work is required.
Since we are working on the 3D SVL, the process includes a coordinate transformation. This process is done by transforming the coordinates from Cartesian to cylindrical and cylindrical to Cartesian. Every time we go back into the main for-loop we add a rotation array (Θ for the cylindrical). This process makes possible the rotation of the periodic unit cell structure and that is what the SVL algorithm is about.

The linear system is solved using PETSc Krylov Methods (KSP) scalable linear equations solvers components. PETSc provides an easy-to-use interface to the combination of a Krylov subspace iterative method and a preconditioner (PC), or a sequential direct solver. KSP uses various Krylov subspace options at runtime via the options database (e.g., -ksp_type gmres, where “gmres” stands for KSP - Generalized Minimal Residual method). We tried different options for solvers. We got our best and fastest result using the Conjugate Gradient (CG) method (e.g., -ksp_type cg) [3].

1: for $m = 0$ to $N_K - 1$ do

2:   $K_x^n \leftarrow K_x[m] : \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}_{M \times 1}$ \hfill $\triangleright$ Orientation Vector in the X-axis

3:   $K_y^n \leftarrow K_y[m] : \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}_{M \times 1}$ \hfill $\triangleright$ Orientation Vector in the Y-axis

4:   $K_z^n \leftarrow K_z[m] : \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}_{M \times 1}$ \hfill $\triangleright$ Orientation Vector in the Z-axis

5:   Solve: Cartesian to Cylindrical ($K_x^n, K_y^n, K_z^n, RHO^n, TH^n$)

6:   $TH^n \leftarrow TH^n + \Theta \quad \triangleright$ Rotation

7:   $K_z^n \leftarrow K_z^n$

8:   $RHO^n \leftarrow RHO^n_{PER} \quad \triangleright$ Lattice Spacing

9:   Solve: Cylindrical to Cartesian ($K_x^n, K_y^n, K_z^n, RHO^n, TH^n$)

10:  $f_{vec}^m \leftarrow \begin{bmatrix} K_x^n \\ K_y^n \\ K_z^n \end{bmatrix}_{3M \times 1}$ \hfill $\triangleright$ Define Right Hand Side $f$

11: procedure SOLVE LINEAR SYSTEM

12:    MatMultTranspose($G, f_{vec}, B$) \hfill $\triangleright$ Matrix-vector multiplication $B = G^T f_{vec}$

13:    MatTransposeMatMult($G, G, &C$) \hfill $\triangleright$ Matrix-Matrix multiplication $C = G^T G$

14:    KSPCreate(&ksp) \hfill $\triangleright$ Create linear solver context

15:    KSPSetOperators(ksp, C, C) \hfill $\triangleright$ Set operators

16:    KSPSetFromOptions(ksp) \hfill $\triangleright$ Set runtime options

17:    KSPSolve(ksp, B, $\Phi^m$) \hfill $\triangleright$ Solve linear system

18: end procedure

19: $S^m \leftarrow AMPN[m] \exp(i \Phi^m)$

20: $U_C^m \leftarrow U_C^m + S^m$

21: end for

22: return $\Phi^m, S^m, U_C^m$
Chapter 3

Improving the Efficiency of the SVL Algorithm

3.1 Motivation

The SVL program solves an overdetermined linear system (see equation 3.1) inside the for-loop using a direct or iterative method for solving linear systems of equations. For example, in the 3D case, the linear system looks as follows:

\[
\begin{bmatrix}
D_x \\
D_y \\
D_z
\end{bmatrix}
\Phi =
\begin{bmatrix}
K_x \\
K_y \\
K_z
\end{bmatrix}
\]

Let us call the grating matrix \( G \) and the grating array \( K \).

\[
G_{3M \times M} \Phi_{M \times 1} = K_{3M \times 1}
\]

\[
(G^T G)_{M \times M} \Phi_{M \times 1} = (G^T K)_{M \times 1}
\]

\[
\Phi_{M \times 1} = (G^T G)_{M \times M} (G^T K)_{M \times 1}
\]  

(3.1)

This allowed us to recognize two bottlenecks on which we could possibly work to improve the code performance.

1. The for-loop runs over the **number of harmonics** stored in the truncated array \( \text{AMNP} \).

2. A large, sparse linear system is solved in each execution of the for-loop.
3.2 Bottlenecks Description

3.2.1 Number of Harmonics

The number of for-loop cycles depends on the number of FFT harmonics stored in the truncated array. Therefore the speed and efficiency of the algorithm can be greatly improved by minimizing the number of times the overdetermined linear system has to be solved inside the loop.

- For the truncated array $AMN$ in the 2D model, the number of FFT harmonics stored in the array is $(11 \times 11)$. Therefore the product of the rows and columns gives us the number of times (121-loops) to complete the full cycle (See Figure 3.1).

$$11 \times 11 = 121$$

- For the truncated array $AMNP$ in the then 3D model, the number of FFT harmonics stored in the array is $(13 \times 13 \times 13)$. Therefore the product of the rows, columns, and heights gives us the number of times (2187-loops) to complete the full cycle (See Figure 3.2).

$$13 \times 13 \times 13 = 2187$$

We want to be able to recover the original unit cell. A truncation of the spatial harmonics does not make it impossible to recover the original unit cell with fewer spatial harmonics values. Figure 3.3 shows the 2D case of recovering the unit cell as the spatial harmonics are added to reconstruct the unit cell. We also notice that we do not need the full set of harmonics to accomplish a good and reasonable reconstruction of the unit cell. This helps us to pick a reasonable number for the truncated harmonics array [17, 20].

3.2.2 Direct or Iterative Method

We need to decide which method is the best to be used to solve the linear system on a parallel computer. Direct methods are great for dense matrices. They solve the linear
Figure 3.1: 2D FFT array size $256 \times 256$ and truncated array size $11 \times 11$

Figure 3.2: 3D FFT array of size $16 \times 16 \times 16$ and 3D truncated array of size $13 \times 13 \times 13$

Figure 3.3: Consequences of truncating spatial harmonics [18]
system using a fixed sequence of operations. In the absence of rounding errors, direct methods would deliver an exact solution like the one we get solving a linear system of equations by Gaussian elimination. In the other hand, iterative methods start with an initial guess and then generate a sequence of improving approximate solutions to the linear system. The \( n^{th} \) iteration is an approximation derived from the \((n - 1)^{th}\) iteration, and iterations continue until satisfying the convergence criteria. Iterative methods are often used for solving linear systems that involve a large number of variables (sometimes on the order of millions), where direct methods would be prohibitively expensive and in some cases impossible even with the best available computer power [8, 9, 21].

In the case of a dense matrix, the operations cost is on the order of \( \frac{2}{3}n^3 \) operations (\( n \) is the matrix dimension), which is too large for most large linear systems. With iterative methods, the cost can be reduced to the order of \( n^2 \) operations for each iteration. The number of iterations that are required to converge within a prescribed tolerance is either independent of \( n \) or scales sublinearly with respect to \( n \) [8, 9, 21].

In our case, the equation (3.1) presents a large sparse matrix \( G \), as discussed above. Direct methods may be inefficient due to the dramatic fill-in, even though extremely efficient direct solvers can be devised on sparse matrices featuring special structures. PETSc offers a wide variety of Krylov subspace iterative methods to be used. In our case, we pick the conjugate gradient method which fits very well our need to solve linear system for the 2D and 3D SVL algorithms.

### 3.3 Improvements Implemented

The number of times the loop is executed depends directly on the number of spatial harmonic stored in the truncation array \( AMNP \). The improvement is done mainly by reducing the number of harmonics. The size of the truncated array is \( NK = NM \times NN \times NP \) and the size indicates the total number of spatial harmonics stored in the array.

Elimination of the gratings according to their amplitude is the only improvement of
the 2D and 3D SVL code that is realized in this research [17, 20].

3.3.1 Elimination of the Gratings According to Their Amplitude

From the truncated array AMNP, we selected the highest spatial harmonics by cutting off the lowest spatial harmonics. This can be done by analyzing the amplitude of each of the spatial harmonic stored in the truncated array and selecting the highest one as our threshold AMNP$_\text{threshold}$. Next, we use the selected threshold amplitude AMNP$_\text{threshold}$ conditional operator to be compared against each spatial harmonic from the truncated array. Each spatial harmonic that is above the threshold condition is retained and the ones that are below the threshold are cleared out from memory. This allows us to build a mask array $K_{\text{AMNP}}$ of 1’s and 0’s [17, 20].

$$|\text{AMNP}| < \text{AMNP}_{\text{threshold}} \quad \text{AMNP}_{\text{threshold}} \approx 0.02$$

Each spatial harmonic with magnitude greater than the threshold has an associated grating value in the grating array $\mathbf{K} = (K_x, K_y, K_z)$. The values in the grating and truncated array work together. So, we perform an array multiplication between the mask array $K_{\text{AMNP}}$ and the grating array $(K_x, K_y, K_z)$, as well as the mask array $K_{\text{AMNP}}$ and the truncated array AMNP. The resulting operation gives us two new arrays $\tilde{K}$ and $\tilde{\text{AMNP}}$ with no zero values that can be stored and 0’s that can be flushed out from memory. The size of the truncated array is now $NK \ll NM \times NN \times NP$.

$$\tilde{K} = K_{\text{AMNP}} \cdot \mathbf{K}$$

$$\tilde{\text{AMNP}} = K_{\text{AMNP}} \cdot \text{AMNP}$$

The threshold that works in many cases is usually around 2% of the maximum harmonic. This procedure affects the shape of the final lattice array, but it can be tolerated considering we are significantly improving the algorithm’s speed. A previous work using this improvement achieved 60 – 90% reduction in the number of planar gratings without significantly distorting the unit cell [17, 20].
• For the truncated array in the 2D model, the number of FFT harmonic stored were $(11 \times 11)$. This means that the for-loop will complete its full cycles after

$$11 \times 11 = 121 \text{ for-loop executions}$$

Eliminating spatial harmonics according to their amplitude reduces the number of harmonics to

$$85 \text{ for-loops executions}$$

• For the truncated array on the 3D model the number of FFT harmonic store were $(13 \times 13 \times 13)$. This means that the for-loop will complete its full cycles after

$$13 \times 13 \times 13 = 2187 \text{ for-loop executions}$$

Eliminating the spatial harmonics according to their amplitude reduces the number of harmonics to

$$735 \text{ for-loop executions}$$

The mask array for the 2D SVL algorithm is shown in Figure 3.4.
Figure 3.4: Mask array $K_{AMN}$: Elimination of gratings according to their amplitude
Chapter 4

Results and Analysis

In this chapter, we show the results of executing the 3D SVL code on TACC-Stampede2. Stampede2 is a supercomputer at the Texas Advanced Computing Center (TACC), the University of Texas at Austin. We first describe the two Stampede2 node architectures, Intel Skylake and Intel Knights Landing, and how we compiled the SVL code for these architectures. We explain how to scale up the problem size while increasing the number of processors using a technique called isoefficiency scaling. We show the runtimes we achieved using these problem sizes. We characterize both node architectures using the Roofline performance modeling methodology, and we evaluate the runtime performance results with respect to the Roofline model.

4.1 TACC Stampede 2 - Technical Specifications

Phase 1 Stampede2 has the second generation of processors based on Intel’s Many Integrated Core (MIC) architecture, 4,200 Knights Landing (KNL) nodes. Stampede2 KNL is not a coprocessor, each 68-core KNL is a stand-alone, self-booting processor that is the sole processor in its node. Phase 2 Stampede2 added a total of 1,736 Intel Xeon Skylake (SKX) nodes [1]. Tables 4.1 and 4.2 show the specifications of the Stampede2 nodes and cores.
Table 4.1: Stampede2-Knights Landing (KNL) node specifications (Phase 1)[1]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model:</strong></td>
<td>Intel Xeon Phi 7250 (“Knights Landing”)</td>
</tr>
<tr>
<td><strong>Total cores per KNL node:</strong></td>
<td>68 cores on a single socket</td>
</tr>
<tr>
<td><strong>Hardware threads per core:</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>Hardware threads per node:</strong></td>
<td>68 x 4 = 272</td>
</tr>
<tr>
<td><strong>clock rate:</strong></td>
<td>1.4 Ghz</td>
</tr>
<tr>
<td><strong>RAM:</strong></td>
<td>96GB DDR4 plus 16GB high-speed MCDRAM</td>
</tr>
<tr>
<td><strong>Cache:</strong></td>
<td>32KB L1 data cache per core; 1MB L2 per two-core tile. In default config, MCDRAM operates as 16GB direct-mapped L3.</td>
</tr>
<tr>
<td><strong>Local storage:</strong></td>
<td>All but 504 KNL nodes have a 107GB /tmp partition on a 200GB Solid State Drive (SSD). The 504 KNLs originally installed as the Stampede1 KNL sub-system each have a 32GB /tmp partition on 112GB SSDs. The latter nodes currently make up the development and flat-quadrant queues.</td>
</tr>
<tr>
<td><strong>Nodes</strong></td>
<td>Stampede2 hosts 4,200 KNL compute nodes</td>
</tr>
</tbody>
</table>
Table 4.2: Stampede2-Skylake (SKX) node specifications (Phase 2)[1]

<table>
<thead>
<tr>
<th>Model:</th>
<th>Intel Xeon Platinum 8160 (&quot;Skylake&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total cores per SKX node:</td>
<td>48 cores on two socket (24 cores/socket)</td>
</tr>
<tr>
<td>Hardware threads per core:</td>
<td>2</td>
</tr>
<tr>
<td>Hardware threads per node:</td>
<td>48 x 2 = 96</td>
</tr>
<tr>
<td>Clock rate:</td>
<td>2.1GHz nominal (1.4-3.7GHz depending on instruction set and number of active cores)</td>
</tr>
<tr>
<td>RAM:</td>
<td>192GB (2.67GHz)</td>
</tr>
<tr>
<td>Cache:</td>
<td>32KB L1 data cache per core; 1MB L2 per core; 33MB L3 per socket. Each socket can cache up to 57MB (sum of L2 and L3 capacity).</td>
</tr>
<tr>
<td>Local storage:</td>
<td>144GB /tmp partition on a 200GB SSD</td>
</tr>
<tr>
<td>Nodes:</td>
<td>Stampede2 hosts 1,736 SKX compute nodes</td>
</tr>
</tbody>
</table>

4.1.1 KNL and SKX Architecture Details

- According to the Knights Landing (KNL) scheme, the cores are grouped in pairs. Each pair of cores occupies a tile with CPU (hardware thread) numbers 0-67 spread across the 68 cores, 1 thread per core on each KNL node. Each core can run up to 4 hardware threads. Each node has 34 active tiles connected by a two-dimensional mesh and each active tile shares a 1MB L2 cache.
### Table 4.3: Target architecture option descriptions[1]

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
<td>Switch target base architecture (instruction set). It must run on all targeted processors.</td>
</tr>
<tr>
<td>MIC-AVX512</td>
<td>Target KNL-specific subset of Intel’s Advanced Vector Extensions 512-bit instruction.</td>
</tr>
<tr>
<td>CORE-AVX512</td>
<td>Target SKX-specific subset</td>
</tr>
<tr>
<td>CORE-AVX2</td>
<td>Target native older Broadwell processors supported on both KNL and SKX.</td>
</tr>
<tr>
<td>-ax</td>
<td>Comma-separated list of alternate instruction sets CORE-AVX512 for SKX, and MIC-AVX512 for KNL.</td>
</tr>
</tbody>
</table>

- Each KNL core has 2 DDR memory controllers with 3 channels, a local L1 cache (32KB data, 32KB instruction) and two 512-bit vector units. Both vector units can execute AVX512 instructions[1] (See Table 4.1).

- According to the Skylake scheme (SKX), CPU (hardware thread) numbers 0-47 are spread across the 48 cores, 1 thread per core, and each core can run up to 2 hardware threads[1](See Table 4.2).

- The command monitoring job status `showq` reports total nodes associated with a job rather than cores, tasks, or hardware threads. The operating system (OS) sees each of a KNL node’s 272 hardware threads and each of a SKX node’s 96 hardware threads as a processor[1] (See Tables 4.1 and 4.2).

### 4.2 Building the SVL Code

The SVL code was compiled on Stampede2 by specifying the target architecture using the following build options: `-xMIC-AVX512` for KNL and `-xCORE-AVX512` for SKX [1] (See Table 4.3).
We also specify an optimization level by using the -O flag. We use the Intel C compiler (icc) to build the executable since it is the default C compiler on Stampede2 [1].

- To build the code to run on the KNL architecture, we use the following compile command:

```bash
$ icc -xMIC-AVX512 -O3 my_program.c -o my_program
```

- To build the code to run on the SKX architecture, we use the following compile command:

```bash
$ icc -xCORE-AVX512 -O3 my_program.c -o my_program
```

### 4.3 Linear System Solver

The sparse linear system to be solved is overdetermined since the number of equations (rows) exceeds the number of variables (columns). For simplicity, we call the right-hand matrix $K$ and the left-hand matrix $G$. We solve the sparse linear system using the least squares method by multiplying each side by the transpose matrix $G^T$ as follows:

$$G\Phi = K$$

$$(G^T G)\Phi = (G^T K)$$

The resulting matrix $(G^T G)$ is a square sparse symmetric positive definite matrix and $(G^T K)$ is a column vector. This linear system is solved using a Krylov Subspace (KSP) method, in particular the Preconditioned Conjugate Gradient (PCG) iterative method (KSPCG) from the PETSc package. The KSP scalable linear equation solvers provide an interface to the combination of a Krylov subspace iterative method and a preconditioner. The Krylov subspace solver can be selected at runtime using the PETSc option `-ksp_type` (e.g., `-ksp_type cg`) [4].
4.4 3D SVL Program

The 3D SVL computer program solves the following sparse linear system inside a for-loop using a Preconditioned Conjugate Gradient method.

\[(G^T G)_{M \times M} \Phi_{M \times 1} = (G^T K)_{M \times 1}\]  

(4.1)

The for-loop cycles depend on the number of FFT harmonics stored in the truncated array AMNP. In our case, we used a truncated array AMNP of size \((13 \times 13 \times 13)\). This means that the for-loop will complete its full cycles after

\[13 \times 13 \times 13 = 2187\] for loop executions.

To reduce the number of loop cycles we introduce an improvement called **Elimination of gratings according to their amplitude**, and take the highest harmonics that passed an established threshold, which reduces the number of harmonics to

\[735\] for-loop executions.

4.5 Isoefficiency Scalability

It should be possible to keep the parallel efficiency \(E\) fixed by increasing both the size of the array in the equation (4.1) and the number of processing elements \(p\) simultaneously while maintaining the growth ratio. The parallel efficiency is defined as

\[E = \frac{S}{p} = \frac{T_{seq}(N)}{p \cdot T_{par}(N, p)}\]

where \(N\) is the problem size, \(p\) is the number of processors, \(T_{seq}(N)\) is the execution time of the sequential algorithm, and \(T_{par}(N, p)\) is the execution time of the parallel algorithm with \(p\) processors.

Isoefficiency analysis allows us to calculate at what rate we should increase the problem size while increasing the number of processors in order to keep the efficiency fixed.
The reason we cannot simply double the problem size when we double the number of processors is because of parallel overhead (for example, communication overhead), which typically depends on both the problem size and the number of processors. Parallel overhead is defined as

\[ T_{\text{over}}(N, p) = pT_{\text{par}}(N, p) - T_{\text{seq}}(N) \]

The parallel efficiency can be expressed as a function of the total overhead \( T_{\text{over}} \) and the sequential execution time \( T_{\text{seq}} \) as follows:

\[ E = \frac{S}{p} = \frac{T_{\text{seq}}(N)}{pT_{\text{par}}(N, p)} = \frac{1}{1 + \frac{T_{\text{over}}(N, p)}{T_{\text{seq}}(N)}} \]

Solving for \( T_{\text{seq}} \), we obtain the isoefficiency equation

\[ T_{\text{seq}}(N) = \frac{E}{1 - E}T_{\text{over}}(N, p) \]

where \( E \) is the desired efficiency to be maintained. Substituting \( W = T_{\text{seq}}(N) \), where \( W \) represents the workload, and, given an expression for \( T_{\text{over}} \), we can solve for \( W \) in terms of \( p \) to give a function that tells us how we should scale up \( W \) as we increase \( p \).

The Conjugate Gradient algorithm is analyzed in [23] and the isoefficiency equation is derived to be

\[ N = \frac{E}{c(1 - E)t_a} \left[ 8p(t_s + t_h) + 4(t_s + t_w + 2t_h)p^{3/2} \right] = \Theta(p^{3/2}) \]

where \( N \) is the problem of size, \( p \) is the number of processors, \( T_{\text{seq}}(N) \) is the execution time of the sequential algorithm, and \( T_{\text{par}}(N, p) \) execution time of the parallel algorithm with \( p \) processors.

Table 4.4 shows the cases for which we increase the array dimension \( M \) from equation (4.1), as we increase the size of the problem and the resources using isoefficiency scalability analysis to estimate the array size \( N = \Theta(p^{3/2}) \)[23].
Let \( N_i \) be the problem size assigned to \( p_i \) cores and \( N_j \) the problem size assigned to \( p_j \) cores. Then

\[
\frac{N_j}{N_i} \propto \left( \frac{p_j}{p_i} \right)^{3/2} = \left( \frac{p_j}{p_i} \right)^{3/2} \]

For example: let \( p_1 = 1 \) and \( p_2 = 2 \)

\[
N_2 \propto 2^{3/2} N_1 = 2.83 N_1
\]

For example: let \( p_1 = 1 \) and \( p_4 = 4 \)

\[
N_4 \propto 4^{3/2} N_1 = 8 N_1
\]

or \( p_2 = 2 \) and \( p_4 = 4 \)

\[
N_4 \propto 2^{3/2} N_2 = 2.83 N_2
\]

There are two parameters we can use to increase the size of the 3D SVL problem: number of grid cells \((N_{P_x} \times N_{P_y} \times N_{P_z})\) and grid resolution \(N_{GP}\). These numbers are used to increase the size of the linear system solved inside a for-loop over the fixed number of harmonics values stored in the truncated array \(AM_{NP}\).

Table 4.4 shows the initial problem size using the isoefficiency formula \(2^{3/2} M^2\). For example, a lattice with \((N_{P_x} \times N_{P_y} \times N_{P_z})= (7 \times 7 \times 7)\) grid cells and grid resolution \(N_{GP}=5\) needs a \(G\) operator of size \(35^3 \times 35^3 = 1.84 \times 10^9\). To scale up the problem, we increase the problem size using the isoefficiency formula \(2^{3/2}(1.84 \times 10^9) = 5.2 \times 10^9\) as we double the number of cores \(p\) (resources), as is shown in Table 4.5 for the \(7 \times 7 \times 7\) lattice. Following the same procedure, every time we increase the problem size, we multiply the previous number by the factor \(2^{3/2}\) and we double the number of cores (resources).

Tables 4.5, 4.6, 4.7 and 4.8 show how we derive the grid size and grid resolution needed to scale up the problem size for isoefficiency scaling for the different lattice sizes, respectively.
Table 4.4: Isoefficiency scalability initial grid size

<table>
<thead>
<tr>
<th>Grid Cells</th>
<th>Grid Resolution</th>
<th>Grid size</th>
<th>Operator size</th>
<th>$2^{3/2}M^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NPx × NPy × NPz)</td>
<td>NGP</td>
<td>(M × M)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(7 × 7 × 7)</td>
<td>5</td>
<td>(35 × 35 × 35)</td>
<td>$35^3 × 35^3$</td>
<td>$1.84 × 10^9$</td>
</tr>
<tr>
<td>(9 × 9 × 9)</td>
<td>5</td>
<td>(45 × 45 × 45)</td>
<td>$45^3 × 45^3$</td>
<td>$8.3 × 10^9$</td>
</tr>
<tr>
<td>(11 × 11 × 11)</td>
<td>5</td>
<td>(55 × 55 × 55)</td>
<td>$55^3 × 55^3$</td>
<td>$2.7 × 10^{10}$</td>
</tr>
<tr>
<td>(13 × 13 × 13)</td>
<td>5</td>
<td>(65 × 65 × 65)</td>
<td>$65^3 × 65^3$</td>
<td>$7.5 × 10^{10}$</td>
</tr>
</tbody>
</table>

Table 4.5: Using isoefficiency analysis $N = \Theta(p^{3/2})$ to find grid resolution (Res)

<table>
<thead>
<tr>
<th>$p$</th>
<th>$2^{3/2}M^2$</th>
<th>M</th>
<th>$M^{1/3}$ Grid size</th>
<th>$\approx$ (Res)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1838265625.00</td>
<td>42875.00</td>
<td>35.00</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>5199400356.24</td>
<td>72106.87</td>
<td>41.62</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>14706125000.00</td>
<td>121268.81</td>
<td>49.50</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>41595202849.91</td>
<td>203949.02</td>
<td>58.86</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>117649000000.00</td>
<td>343000.00</td>
<td>70.00</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.6: Using isoefficiency analysis $N = \Theta(p^{3/2})$ to find grid resolution (Res)

<table>
<thead>
<tr>
<th>$p$</th>
<th>$2^{3/2}M^2$</th>
<th>M</th>
<th>$M^{1/3}$ Grid size</th>
<th>$\approx$ (Res)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8303765625.00</td>
<td>91125.00</td>
<td>45.00</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>23486595931.29</td>
<td>153253.37</td>
<td>53.51</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>66430125000.00</td>
<td>257740.42</td>
<td>63.64</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>187892767450.28</td>
<td>433465.99</td>
<td>75.68</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>531441000000.00</td>
<td>729000.00</td>
<td>90.00</td>
<td>10</td>
</tr>
</tbody>
</table>
Table 4.7: Using isoefficiency analysis $N = \Theta(p^{3/2})$ to find grid resolution ($\text{Res}$)

<table>
<thead>
<tr>
<th>$p$</th>
<th>$2^{3/2}M^2$</th>
<th>M</th>
<th>$M^{1/3}$ Grid size</th>
<th>$\approx$ (Res)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27680640625.00</td>
<td>166375.00</td>
<td>55.00</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>78292674774.10</td>
<td>279808.28</td>
<td>65.41</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>221445125000.00</td>
<td>470579.56</td>
<td>77.78</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>626341398192.81</td>
<td>791417.34</td>
<td>92.50</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>1771561000000.00</td>
<td>1331000.00</td>
<td>110.00</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.8: Using isoefficiency analysis $N = \Theta(p^{3/2})$ to find grid resolution ($\text{Res}$)

<table>
<thead>
<tr>
<th>$p$</th>
<th>$2^{3/2}M^2$</th>
<th>M</th>
<th>$M^{1/3}$ Grid size</th>
<th>$\approx$ (Res)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75418890625.00</td>
<td>274625.00</td>
<td>65.00</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>213316835962.02</td>
<td>461862.36</td>
<td>77.30</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>603351125000.00</td>
<td>776756.80</td>
<td>91.92</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1706534687696.13</td>
<td>1306344.02</td>
<td>109.32</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>4826809000000.00</td>
<td>2197000.00</td>
<td>130.00</td>
<td>10</td>
</tr>
</tbody>
</table>
4.6 Stampede2 Performance Results

The tables in this section show the run times and parallel efficiencies for the lattice array sizes \((7 \times 7 \times 7), (9 \times 9 \times 9), (11 \times 11 \times 11), \) and \((13 \times 13 \times 13)\) with different grid resolutions as a way to increase the problem sizes. In each table, the columns show how the run time decreases as we increase the number of cores \(p\). The rows show how the run time increases as we increase the problem size. With perfect isoefficiency scaling – that is, maintaining the same efficiency as we scale up the problem, we should see an efficiency of one along the diagonals of the efficiency tables.

4.6.1 Lattice \(7 \times 7 \times 7\) - KNL vs. SKX

In this subsection, we compare the run time performance of the two different architectures (KNL vs. SKX) on Stampede2 using one node. The plot (See Figure 4.2) and table for SKX (See Table 4.11) show that it executes the \((7 \times 7 \times 7)\) SVL lattice in less time than KNL (See Figure 4.1 and Table 4.9). Also, SKX allows us to reach higher resolution number than KNL before it requires the use of two nodes.

KNL

In Table 4.9 and Figure 4.1, we see that the run time for each of the problem sizes decreases as we increase the number of cores. However, as we look along the diagonal in Table 4.10, we see that the parallel efficiency decreases from 1.00 to 0.42, rather than remaining constant.
Figure 4.1: Execution times on KNL for $7 \times 7 \times 7$ lattice, with 735 harmonics

Table 4.9: Execution time (in seconds) on KNL for $7 \times 7 \times 7$ lattice as a function of the array size and number of processing elements $p$

<table>
<thead>
<tr>
<th>$p$</th>
<th>Grid Resolution for a $(7 \times 7 \times 7)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>418.78</td>
</tr>
<tr>
<td>2</td>
<td>333.58</td>
</tr>
<tr>
<td>4</td>
<td>209.84</td>
</tr>
<tr>
<td>8</td>
<td>153.71</td>
</tr>
<tr>
<td>16</td>
<td>124.94</td>
</tr>
</tbody>
</table>
Table 4.10: Efficiency on KNL for $7 \times 7 \times 7$ lattice as a function of array size

<table>
<thead>
<tr>
<th>$p$</th>
<th>Grid Resolution for $a (7 \times 7 \times 7)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.63</td>
</tr>
<tr>
<td>4</td>
<td>0.50</td>
</tr>
<tr>
<td>8</td>
<td>0.34</td>
</tr>
<tr>
<td>16</td>
<td>0.21</td>
</tr>
</tbody>
</table>

SKX

In Table 4.11 and Figure 4.2, we again see that the run time for each of the problem sizes decreases as we increase the number of cores. Comparing the run times with those for KNL, we see that the SKX run times are three to eight times faster, with a smaller difference between the architectures for the larger problem sizes. As we look along the diagonal in Table 4.12, we see that the parallel efficiency decreases from 1.00 to 0.29, but at about the same rate as for KNL for the same problem sizes.

Table 4.11: Execution time (seconds) on SKX for $7 \times 7 \times 7$ lattice, as a function of the array size and number of processing elements $p$

<table>
<thead>
<tr>
<th>$p$</th>
<th>Grid Resolution for $a (7 \times 7 \times 7)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>50.54</td>
</tr>
<tr>
<td>2</td>
<td>41.67</td>
</tr>
<tr>
<td>4</td>
<td>28.01</td>
</tr>
<tr>
<td>8</td>
<td>21.46</td>
</tr>
<tr>
<td>16</td>
<td>18.15</td>
</tr>
</tbody>
</table>
Figure 4.2: Execution times on SKX for $7 \times 7 \times 7$ lattice, with 735 harmonics

Table 4.12: Efficiency on SKX for $7 \times 7 \times 7$ lattice as a function of array size

<table>
<thead>
<tr>
<th>P</th>
<th>Grid Resolution for $a (7 \times 7 \times 7)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.61</td>
</tr>
<tr>
<td>4</td>
<td>0.45</td>
</tr>
<tr>
<td>8</td>
<td>0.29</td>
</tr>
<tr>
<td>16</td>
<td>0.17</td>
</tr>
</tbody>
</table>
4.6.2 SKX Results

We use the Skylake (SKX) architecture for the remaining lattice sizes since it has better performance.

**Lattice 9 × 9 × 9**

In this subsection, we show the run time performance for the (9 × 9 × 9) lattice (See Figure 4.3). The problems with resolution 5, 6 and 7 run on one node. The problem with resolution 8 needs to be executed using two nodes. Resolution values higher than 7 made the program stop before reaching the completion of the loop 735, possibly due to running out of memory (See Tables 4.13 and 4.14).

![Figure 4.3: Execution times on SKX for 9 × 9 × 9 lattice, with 735 harmonics](image)
Table 4.13: Execution time (seconds) on SKX for $9 \times 9 \times 9$ lattice as a function of the array size and number of processing elements $p$

<table>
<thead>
<tr>
<th>$p$</th>
<th>Grid Resolution for a $(9 \times 9 \times 9)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>131.13</td>
</tr>
<tr>
<td>2</td>
<td>97.08</td>
</tr>
<tr>
<td>4</td>
<td>61.37</td>
</tr>
<tr>
<td>8</td>
<td>48.61</td>
</tr>
<tr>
<td>16</td>
<td>36.77</td>
</tr>
</tbody>
</table>

Table 4.14: Efficiency on SKX for $9 \times 9 \times 9$ lattice as a function of array size

<table>
<thead>
<tr>
<th>$p$</th>
<th>Grid Resolution for a $(9 \times 9 \times 9)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.68</td>
</tr>
<tr>
<td>4</td>
<td>0.53</td>
</tr>
<tr>
<td>8</td>
<td>0.34</td>
</tr>
<tr>
<td>16</td>
<td>0.22</td>
</tr>
</tbody>
</table>
**Lattice** $11 \times 11 \times 11$

In this subsection, we show the run time performance for the $(11 \times 11 \times 11)$ lattice (See Figure 4.4 and Table 4.15). The problems with resolution 5, and 6 run on one node. The problem with resolution 7 needs to be executed using two nodes. Resolution values higher than 6 made the program stop before reaching the completion of loop 735, possibly due to running out of memory.

![Graph showing execution times on SKX for $11 \times 11 \times 11$ lattice, with 735 harmonics](image)

Figure 4.4: Execution times on SKX for $11 \times 11 \times 11$ lattice, with 735 harmonics
Table 4.15: Execution time (seconds) on SKX for $11 \times 11 \times 11$ lattice as a function of the array size and number of processing elements $p$

<table>
<thead>
<tr>
<th>$p$</th>
<th>Grid Resolution for a $(11 \times 11 \times 11)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>289.21</td>
</tr>
<tr>
<td>2</td>
<td>201.47</td>
</tr>
<tr>
<td>4</td>
<td>125.99</td>
</tr>
<tr>
<td>8</td>
<td>89.82</td>
</tr>
<tr>
<td>16</td>
<td>138.10</td>
</tr>
</tbody>
</table>

Table 4.16: Efficiency on SKX for $11 \times 11 \times 11$ lattice, as a function of array size

<table>
<thead>
<tr>
<th>$p$</th>
<th>Grid Resolution for a $(11 \times 11 \times 11)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.72</td>
</tr>
<tr>
<td>4</td>
<td>0.57</td>
</tr>
<tr>
<td>8</td>
<td>0.40</td>
</tr>
<tr>
<td>16</td>
<td>0.28</td>
</tr>
</tbody>
</table>
**Lattice $13 \times 13 \times 13$**

In this subsection, we show the run time performances for the $(13 \times 13 \times 13)$ lattice (See Figure 4.5). The problems with resolution 5 run in one node. The problem with resolution 6 needs to be executed using two nodes. Increasing the resolution for values higher than 6 made the program stop before the loop 735, possibly due to running out of memory (See Table 4.17 and 4.18).

![Figure 4.5: Execution times on SKX for $13 \times 13 \times 13$ lattice, with 735 harmonics](image-url)
Table 4.17: Execution time (seconds) on SKX for $13 \times 13 \times 13$ lattice, as a function of the array size and number of processing elements $p$

<table>
<thead>
<tr>
<th>$p$</th>
<th>Grid Resolution for a $(13 \times 13 \times 13)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>561.96            1244.03</td>
</tr>
<tr>
<td>2</td>
<td>390.68            819.91</td>
</tr>
<tr>
<td>4</td>
<td>233.19            447.23</td>
</tr>
<tr>
<td>8</td>
<td>155.75            310.78</td>
</tr>
</tbody>
</table>

Table 4.18: Efficiency on SKX for $(13 \times 13 \times 13)$ lattice as a function of array size

<table>
<thead>
<tr>
<th>$p$</th>
<th>Grid Resolution for a $13 \times 13 \times 13$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1.00                                          1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.72                                          0.76</td>
</tr>
<tr>
<td>4</td>
<td>0.60                                          0.70</td>
</tr>
<tr>
<td>8</td>
<td>0.45                                          0.50</td>
</tr>
</tbody>
</table>

90
4.7 Discussion of Stampede2 Performance Results

The execution times for multiple runs of the 3D SVL code on SKX Stampede2 architecture (See Tables 4.11, 4.13, 4.15, and 4.17) show that SKX has faster run times as we increase the number of cores $p$. The plots (See Figures 4.2, 4.3, 4.4, and 4.5) show how the time decreases as we increase the number of cores. The efficiency tables 4.12, 4.14, 4.16 and 4.18) show how the efficiency decreases as we increase the number of cores which is due to an increase in parallel overheads. However, as we increase the lattice size, the efficiency improves. Therefore, we expect that the parallel efficiency should improve on a much bigger lattice problem size.

4.8 The Roofline Model

The Roofline performance model provides a framework for evaluating the performance of a code on a given architecture and across architectures [22]. New multi-core processor architectures such as Intel’s Knights Landing (KNL) and Intel Skylake (SKX) provide multiple unique features. Therefore it is important to understand the inherent hardware limitations and architectural bottlenecks of each architecture.

The following definitions are used by the Roofline model:

1. $P_{max}$ = Achievable peak performance of a loop (Flops(work) per second [Flop/s]), assuming that data comes from $L1$ cache (this is not necessarily $P_{peak}$)

2. $I$ = Arithmetic Intensity (Flops (work) per byte transferred [F/B]) over the slowest data path utilized (the bottleneck).

3. Machine Balance Point ([F/B])
   $$x = \frac{P_{max}}{b_S}$$

4. $b_S$ = Achievable peak bandwidth (byte per second transferred [B/s]) of the slowest data path utilized
5. Expected performance

\[ P = \min(P_{\text{max}}, I \cdot b_S) \]

So, in other words, if the arithmetic intensity \( I \) is less than the machine balance point \( x \), then the code will be memory-bound and limited by the memory bandwidth, rather than by the peak floating point performance.

4.8.1 Conjugate Gradient Method - Sparse Matrix-Vector (SpMV) product

The CG algorithm detailed below is one of the best iterative methods for solving a symmetric positive definite linear system of the form \( Ax = b \), where \( A \) is a sparse matrix, \( b \) is the independent term, and \( x \) is the unknown vector. The input vector \( x_0 \) is the initial guess, and it can be an approximate initial solution or 0. The algorithm 7 describes the CG method pseudocode, and it also provides an estimation of the FLOPS in the SpMV, DOT and AXPY columns. Table 4.19 shows the number of FLOPs per line [7]. The total number of FLOPs for the SpMV operation is

\[ \text{OPS}_{\text{SpMV}} = 2EE \]

The total number of FLOPs of the CG algorithm can be calculated by adding the FLOPs shown in Table 4.19.

\[ \text{OPS}_{\text{CG}} = 2EE + 5N + NIT(2EE + 12N) \]

where \( \text{NNZ} \) is the number of nonzero elements, \( EE \) is the number of stored elements by the format to reproduce the matrix, \( NIT \) is the number of complete iterations performed and \( N \) is the order of the matrix, \( N = N\text{ROWS} \) for square matrices [7, 11].

We use the above analysis to estimate the number of FLOPS for executions of the SVL code for different problem instances.
Algorithm 7 Conjugate Gradient

1: $r_0 := b - Ax_0$ \hspace{1cm} ▷ Residual
2: $\delta_0 := ||r_0||_2$
3: $p_0 := r_0$ \hspace{1cm} ▷ Initial direction
4: $\gamma_0 := (r_0^T r_0)$
5: while $\delta_{k+1}$ is larger than convergence criteria do
6: \hspace{1cm} $q_k := Ap_k$ \hspace{1cm} ▷ Matrix-vector product
7: \hspace{1cm} $\alpha_k := \frac{\gamma_k}{(p_k^T q_k)}$ \hspace{1cm} ▷ Inner product
8: \hspace{1cm} $x_{k+1} := x_k + \alpha_k p_k$ \hspace{1cm} ▷ Vector update
9: \hspace{1cm} $r_{k+1} := r_k - \alpha_k q_k$ \hspace{1cm} ▷ Vector update
10: \hspace{1cm} $\delta_{k+1} := ||r_{k+1}||_2$
11: \hspace{1cm} $\gamma_{k+1} := (r_{k+1}^T r_{k+1})$ \hspace{1cm} ▷ Inner product
12: \hspace{1cm} $\beta_k := \frac{\gamma_{k+1}}{\gamma_k}$
13: \hspace{1cm} $p_{k+1} := r_{k+1} + \beta_k p_k$ \hspace{1cm} ▷ Vector update
14: \hspace{1cm} $k := k + 1$
15: end while
16: The result is $x_{k+1}$
<table>
<thead>
<tr>
<th>Line</th>
<th>SpMV</th>
<th>DOT</th>
<th>AXPY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2EE</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2N</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2N</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2EE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2N</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2N</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>2N</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>2N</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>2N</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>2N</td>
<td></td>
</tr>
</tbody>
</table>
4.8.2 KNL and SKX Specifications

The KNL and SKX technical specifications necessary for the Roofline model are presented in the following tables (See Tables 4.20, 4.21, 4.22 and 4.23). The Roofline model will help to understand the similar and different performance aspect between the KNL and SKX architectures.

**Table 4.20: KNL - Intel Xeon Phi 7250 according with Intel[6, 14]**

<table>
<thead>
<tr>
<th>Xeon Phi Series</th>
<th>sSpec Number</th>
<th>Cores (Threads)</th>
<th>Clock (MHz)</th>
<th>L2 Cache Quantity</th>
<th>MCDRAM Memory</th>
<th>DDR4 Memory</th>
<th>Peak DP Compute</th>
<th>TDP (W)</th>
<th>Socket</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon Phi 7250</td>
<td>SR2MD (B0)</td>
<td>68 (272)</td>
<td>1400</td>
<td>1600</td>
<td>34 MB</td>
<td>16 GB</td>
<td>400+ GB/s</td>
<td>384 GB</td>
<td>102.4 GB/s</td>
<td>3046 GFLOPS 215 230 SVLCLGA3647 June 20, 2016</td>
</tr>
<tr>
<td>SR2X1 (B0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.21: KNL - Intel Xeon Phi 7250 according with TACC-Stampede2**

<table>
<thead>
<tr>
<th>Model</th>
<th>Cores (Threads)</th>
<th>Clock (GHz)</th>
<th>L1 cache per core</th>
<th>L2 per 2-cores tile</th>
<th>L3 (MCDRAM)</th>
<th>DDR4 Memory</th>
<th>High Speed MCDRAM Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Phi 7250</td>
<td>68 (272)</td>
<td>1.4</td>
<td>32 KB</td>
<td>1 MB</td>
<td>16 GB</td>
<td>96 GB</td>
<td>16 GB</td>
</tr>
</tbody>
</table>

**Table 4.22: SKX - Intel Platinum 8160 according with Intel[5, 15]**

<table>
<thead>
<tr>
<th>Xeon Platinum Series</th>
<th>sSpec Number</th>
<th>Cores (Threads)</th>
<th>Clock (GHz)</th>
<th>Turbo Boost All-Core/2.0 (/Max 3.0)</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
<th>TDP (W)</th>
<th>Socket</th>
<th>I/O bus</th>
<th>Memory</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon Platinum 8160</td>
<td>SR3B0 (H0)</td>
<td>24 (48)</td>
<td>2.1</td>
<td>2.8/3.7 GHz</td>
<td>24 MiB</td>
<td>33.00 MiB</td>
<td>150</td>
<td>LGA 3647</td>
<td>3 10.4 GT/s UPI</td>
<td>6 DDR4-2666</td>
<td>July 17, 2017</td>
</tr>
</tbody>
</table>

**Table 4.23: SKX - Intel Platinum 8160 according with TACC-Stampede2**

<table>
<thead>
<tr>
<th>Model</th>
<th>Cores (Threads)</th>
<th>Clock (GHz)</th>
<th>L1 cache per core</th>
<th>L2 per cores</th>
<th>L3 per socket</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Platinum 8160</td>
<td>48 (96)</td>
<td>2.1 (Nominal)</td>
<td>1.4-3.7 (Depending on instruction set and number of active cores)</td>
<td>32 KB</td>
<td>1 MB</td>
<td>33 MB</td>
</tr>
</tbody>
</table>

In terms of double precision (DP) \((1 \text{ Byte (B)} = 8 \text{ bits}, \ 1 \text{ Word } (W) = 1 \text{ B})\)

\[
DP = 64 \text{ bits} = 8 \text{ B} = 8 \text{ W}.
\]
KNL and SKX support 512 bits for instruction. Thus, they can process eight double-precision operations per clock-cycle (DP Flops/cycle)

\[
\frac{512 \text{ bits}}{\text{cycle}} = 8 \frac{\text{DP Flops}}{\text{cycle}}
\]

### 4.9 Case Study using Roofline Model

We would need to measure the number of bytes transferred from memory to determine the arithmetic intensity of the CG in the 3D SVL code. Sparse Matrix-Vector (SpMV) multiplication has low arithmetic intensity because it is bound by memory bandwidth. We were not able to accurately measure memory transfers due to hardware counter limitations on Stampede2. Sparse Matrix-Vector (SpMV) multiplication has low arithmetic intensity because it is bound by memory bandwidth. A research paper that studies the performance of the CG algorithm estimates the arithmetic intensity of CG is around 1 FLOP per 6B [13]. We use this value for \( I \) in our analysis below.

#### 4.9.1 KNL Roofline Model for CG

- Arithmetic Intensity (\( I \))
  \[
  I \approx \frac{1}{6} \frac{\text{Flops}}{B}
  \]

- BandWidth (\( BW \)) MCDRAM (from Stampede2 User Guide)
  \[
  BW = 400 \frac{\text{GB}}{s}
  \]

- BandWidth (\( BW \)) DDR4 (from Stampede2 User Guide)
  \[
  BW = 102.4 \frac{\text{GB}}{s}
  \]
• Peak Performance ($P_{\text{peak}}$)

\[ P_{\text{peak}} = \text{Clock Speed (GHz)} \times 68 \text{ cores} \times \frac{\text{Number of Operations}}{\text{cycle}} \]

\[ P_{\text{peak}} = 1.4 \text{ GHz} \times 68 \times 8 \frac{\text{DP Flops}}{\text{cycle}} = 761.6 \frac{\text{GFlops}}{\text{s}} \]

• Achievable Memory-bound Performance ($P_{MCDRAM}$ and $P_{\text{DDR4}}$)

\[ P_{\text{BW}} = I \cdot b_s \]

\[ P_{MCDRAM} = \frac{1}{6} \frac{\text{Flop}}{B} \times 400 \frac{\text{GB}}{s} = 66.7 \frac{\text{GFlops}}{s} \]

\[ P_{\text{DDR4}} = \frac{1}{6} \frac{\text{Flop}}{B} \times 102.4 \frac{\text{GB}}{s} = 17.07 \frac{\text{GFlops}}{s} \]

• Now applying the roofline model formula

\[ P = \min(P_{\text{max}}, I \cdot b_s) \]

If running out of MCDRAM:

\[ P = \min(761.6, 66.7) \frac{\text{GFlops}}{s} \]

If running out of DRAM:

\[ P = \min(761.6, 17.07) \frac{\text{GFlops}}{s} \]

Because the SVL code is running out of DRAM, due to its large memory requirements, we take 17 GFlops/s as the maximum predicted performance.

4.9.2 SKX Roofline Model for CG

• BandWidth (BW) DDR4 (from Stampede2 User Guide)

\[ BW = 102.4 \frac{\text{GB}}{s} \]
• Peak Performance ($P_{\text{peak}}$)

\[
P_{\text{peak}} = \text{Clock Speed (GHz)} \times 48 \text{ cores} \times \frac{\text{Number of Operations}}{\text{cycle}}
\]

\[
P_{\text{peak}} = 2.1 \text{ GHz} \times 48 \times 8 \frac{\text{DP Flops}}{\text{cycle}} = 806.4 \frac{\text{GFlops}}{s}
\]

• Achievable Memory BandWidth Bound Performance ($P_{\text{DDR4}}$)

\[
P_{\text{BW}} = I \cdot b_s
\]

\[
P_{\text{DDR4}} = \frac{1}{6} \frac{\text{Flop}}{B} \times 102.4 \frac{\text{GB}}{s} = 17.07 \frac{\text{GFlops}}{s}
\]

• Now applying the roofline model formula

\[
P = \min(P_{\text{max}}, I \cdot b_s)
\]

\[
P = \min(806.4, 17.07) \frac{\text{GFlops}}{s}
\]

Thus our predicted maximum performance for CG on SKX is also 17 GFlops/s.

### 4.10 Discussion of Roofline Model Analysis

The resulting times collected after multiple runs of the SVL code on KNL and SKX architectures show that SKX has better performance than KNL. Section 4.6.1 shows the time tables and plots for a periodic structure lattice of $7 \times 7 \times 7$ with multiple resolution grids using one node until two nodes are needed.

Based on the above Roofline model analysis, we would expect similar performance on KNL and SKX. However, performance on KNL is 3-4 times slower for the larger problem sizes. The CG algorithm is memory bound, and KNL has a smaller memory and no L3 cache. KNL is also designed to be used with shared memory programming rather than MPI-only and MPI-only has higher overheads.
4.11 3D SVL Plotting Results

The following depicts the results of the 3D SVL program executed on TACC-Stampede2. The Table 4.24 show a brief description of each of the three variables outputs as a result of executing the SVL program. Figure 4.6 is the lattice grid without spatially variance. Figures 4.7, and 4.8 are the result of performing a cylindrical rotation with respect to one corner of the lattice structure.

Table 4.24: 3D SVL Stampede2 Results

<table>
<thead>
<tr>
<th>Variables</th>
<th>Equation Name</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHI</td>
<td>Linear System Solution</td>
<td>$\Phi = A\backslash f$</td>
</tr>
<tr>
<td>S</td>
<td>Exponential</td>
<td>$S = \varepsilon_m = a_m \exp(j \Phi(\vec{r}))$</td>
</tr>
<tr>
<td>UC</td>
<td>Addition</td>
<td>$UC = UC + S \Rightarrow \varepsilon(\vec{r}) = \sum_{m=1}^{NK} \varepsilon_m$</td>
</tr>
</tbody>
</table>

Figure 4.6: 3D Lattice ($5 \times 5 \times 5$) without SVL
Figure 4.7: 3D Lattice ($5 \times 5 \times 5$) with SVL-Cylindric, $\Phi = A^{-1} f$
Figure 4.8: 3D Lattice ($5 \times 5 \times 5$) with SVL-Cylindric, $UC = UC + S$
Chapter 5

Conclusions and Future Work

5.1 Conclusion

- The main computational bottleneck in the SVL algorithm is calculating the grating phase function $\Phi(\vec{r})$ using the gradient equation.

$$\nabla \Phi(\vec{r}) = \vec{K}(\vec{r})$$

This equation is solved by transforming the gradient equation into the matrix form $Ax = b$, using the finite-difference method so that it can be solved numerically. Direct methods such as LU or QR factorization are possible for small to moderate size problems, but are not feasible for very large problems involving many millions or billions of unknowns. Instead, iterative methods such as Conjugate Gradient must be used.

- The parallel implementation of the SVL algorithm was completed using the Portable Extensible Toolkit for Scientific Computation (PETSc) configured for complex numbers with a C programing language. PETSc was selected for implementing the parallel version for this project because it provides a large variety of matrix formats, including dense and compressed sparse row storage, for sequential and parallel applications. In our case, we used the sparse format matrix, since the matrices built in the code are sparse. Also, it can be used on different computer systems that have the necessary compiler plus libraries for the code.

- Having a faster and portable implementation of the SVL code will enable the sim-
ulation of the metamaterial device to be completed before construction. This is important because it will help the code users to simulate their designs before construction in a short time, helping the user to do the necessary modification on the device before construction and obtain a most reliable product.

- We reduce the computational time by building a mask array $K_{AMNP}$. The improvement requires the Elimination of the Gratings According to their Amplitude by analyzing the amplitudes of the truncated harmonic array $A_{Truncated}$ and selects the highest spatial harmonics amplitudes that are above a threshold $AMNP_{Threshold}$ and eliminates the ones that fall below the threshold.

- PETSC shows limited scalability on the implementation of the SVL algorithm because PETSc is limited by the memory bandwidth and network latency between compute nodes. Also, PETSc is not thread-safe so only can be used with MPI-only code. We expect that using a shared memory implementation would give better performance and scalability.

5.2 Future Work

- The performance of this SVL code could be improved by implementing the following improvements into the code:
  
  - Storing the grating vector and eliminating every other grating vector that is parallel.
  
  - By the combination array multiplication of the two previous improvements.

- There are other parallel tools available that could be used to implement a heterogeneous version of the SVL algorithm. For example, a C library that enables performance and portability across diverse and evolving multicore/manycore architectures (GPUs, Multicore CPUs).
• Results show that the SVL isoefficiency scaling cannot maintain the efficiency as we scale up the problem size and the compute resources simultaneously. On the other hand, the tables show that the SVL code increases its efficiency as the problem size increases. Considering a more significant problem could show improving scalability.

• The lower values we get in the efficiency are due to the parallel overhead. We cannot account for all the overhead as interprocessor communication. We suspect that memory pressure and contention are causing some of the overhead. We can try to reduce that effect by improving the way the linear system is solved.

• The main goals of the future work are
  – Improve the efficiency by reducing communication overhead,
  – Improve scalability (as the problem size and number of processors grow),
  – Show reliability (including error bounds) and portability (across all important parallel machines),
  – Explore use of accelerator plus special purpose hardware,
  – Measure the FLOPs and data transfer to calculate the arithmetic intensity of the SVL algorithm and develop performance optimization based on the roofline model.
References


Curriculum Vitae

Henry R. Moncada López was born on April 30, 1970, in Arequipa, Perú, the first son of Rogelio Moncada Teran and Genobertha López Mendieta. In 1995, he received a bachelor’s degree in physics from the Universidad Nacional Mayor de San Marcos (UNMSM), Lima-Peru. After finishing his studies, he worked in the Geophysical Institute of Peru (IGP), where he became involved with numerical simulations and data analysis of seismic events. It was there that he realized how amazing it is to be able to apply numerical solutions to real life problems and he learned to appreciate the importance of mathematics in other sciences. In 2009, he received a master’s degree in applied mathematics from the Department of Mathematics at the University of New Mexico. In the Fall of 2012, he was accepted into the Computational Science Ph.D. program at the University of Texas at El Paso (UTEP), a program that combines science and computational skills.

In Spring 2013, he started work with Dr. Shirley Moore and Dr. Raymond C. Rumpf to parallelize the Spatially Variant Lattice (SVL) Algorithm. The algorithm exploits directional dependencies by spatially vary the unit cell orientation. In 2012, he joined the SIAM Student Chapter, and by 2013 became the vice-president and by 2014 the president. One significant accomplishment during his time with SIAM was the Friday SIAM Seminar. The SIAM Seminar host students, professors and expert speakers on different topics in science, mathematics, and computer science to the UTEP student community. Henry had the opportunity to attend the Supercomputing Conference (SC) multiple times hosted in different cities around the US. This experience gave him the chance to meet people in his research field, observe the many research branches, and develop expertise in supercomputer performance and modeling.

Permanent address: 414 Virginia Dr.
Oak Ridge, Tennessee 37830