High-Speed bit-loading algorithms for Dynamic Spectrum Management in ADSL

Andrew Bolster

May 23, 2011

Executive Summary

Digital Subscriber Lines (DSL) use advanced bit-loading algorithms to maximise spectral efficiency and throughput, often termed 'Dynamic Spectrum Management" (DSM). Provider industry practise has been to use Level 1 DSM algorithms, where each line individually adjusts its power spectrum against the noise that it can sense. This is called "Waterfilling" and is not optimally efficient in terms of total bundle capacity.

Recent research into more advanced, Level 2, DSM algorithms, which consider the total noise characteristics of the bundle, show near-optimal performance, at a cost of significant computational complexity, making them unuseable in a consumer context.

The evolution of 'classical' fixed function rasterization pipeline Graphic Processing Units (GPU) into fully-programmable devices suitable for General Purpose computing on GPU (GPGPU) opens up these massively parallel floating point processors for practical computation, and introduces the possibility of re-implementing existing algorithms to leverage this new hardware, potentially allowing for near-optimal algorithms to be used in the field.

Acknowledgements

Thanks and praise must go to my project supervisor, Prof. Alan Marshall, whose occasional kick's-in-the-ass stopped me from going too far down the rabbit hole. Thanks also go to Dr Alister McKinley, whose PhD work this project is based on, for his expert domain knowledge, bar tab, and occasional supply runs. Further, I'd like to acknowledge the support and technical guidance of the StackExchange community, and Mr Olan Byrne for his aid in the layout of this document.

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at Queen's University Belfast or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at QUB or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed:_____

Contents

Acronyms

Γ	The Shannon Gap derived from equation (2.18)
α	Fraction of an application that cannot be parallelised
$\Delta p(k)$	Single-bit addition power
$\sigma_n^2(k)$	Background noise power experiences by user n on channel k
$h_{ij}(k)$	XTG from user i to user j on channel k
L_k	The Lagrangian sum of a bitloaded channel
$p_n(k)$	Power provisioned on channel k for user n
ACS	Auto-Configuration Server
ADSL	Asymmetric Digital Subscriber Lines
API	Application Programming Interface
CO	Central Office
CP	Customer Premises
CPE	Customer Premises Equipment
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMT	Discrete Multi-Tone modulation
DSL	Digital Subscriber Lines
DSLAM	DSL Access Multiplexer
DSM	Dynamic Spectrum Management
EMS	Element Management System
FDD	Frequency Division Duplexing
FEXT	Far End Cross-Talk
FM	Fixed-Margin water filling

FTTx Fibre-to-the Home/Building/Cabinet

GPGPU General Purpose computing on GPU

GPU Graphics Processing Units

ISB Iterative Spectrum Balancing

ISP Internet Service Provider

ITU International Telecommunications Union

IWF Iterative Water-Filling

LC Levin-Campello Algorithm

LT Line Termination, at CO side

MIMD Multiple Instruction, Multiple Data Stream

MIPB Multi-user Incremental Power Balancing, aka Greedy

MISD Multiple Instruction, Single Data Stream

MPI Message Passing Interface

NEXT Near End Cross-Talk

NT Network Termination, at CP side

OFDM Orthogonal Frequency-Division Multiplexing

OSB Optimal Spectrum Balancing

POTS Plain Old Telephone Service

PSD Power Spectral Density

PU Processing Unit

PyCUDA CUDA wrapper for the Python programming language

QAM Quadrature Amplitude Modulation

RA Rate Adaptive water filling

RF Radio Frequency

SIMD Single Instruction, Multiple Data Stream

SIMT Single Instruction Multiple Thread Stream

SISD Single Instruction, Single Data Stream

SM Streaming Multiprocessor

SMC Spectrum Maintenance Centre

SNR Signal to Noise Ratio

SP Streaming Processor

SPMD Single Program Multiple Data Stream

SSM Static Spectrum Management

VDMT Vectored DMT

VDSL Very high bit-rate Digital Subscriber Lines

xDSL Catch-all term for variety of DSL systems

XT Cross-Talk

XTG Cross Talk Gain

Chapter 1

Introduction

If there are two characteristics the internet has shown over the past 30 years, it's spacial growth, and an insatiable demand for bandwidth. By December 2010, almost 30% of the world's population was connected to the Internet. In the developing world, much of this can be accredited to the growth of mobile connectivity, but even for DSL, a technology that dates back to 1984, growth continues. In Q3 2010, BT announced that it had added nearly 200,000 DSL lines in the UK, bringing their total market share to 53%. [1]

The rise of applications including live-video streaming such as BBC iPlayer, distributed content systems such as BitTorrent, and real-time gaming platforms such as XBox Live, continuously pressure Internet Service Providers (ISP's) to improve backbone and last-mile infrastructure to satisfy customer expectations. Technologies are coming to market, such as Fibre to the Home/Building/Cabinet (FTTx) which involves the replacement of copper lines with optical fibres, but in many cases, this is expensive and will not be done outside of dense municipal areas.

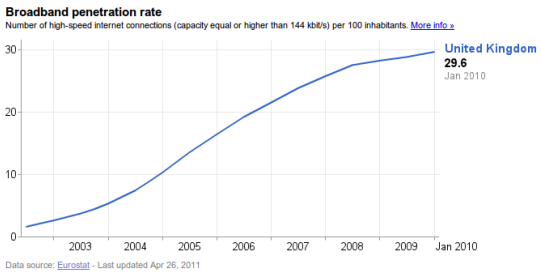


Figure 1: Growth in Broadband connections in the UK

For other users, Digital Subscriber Lines (DSL) built on top of the existing plain old telephone system (POTS) are often the only option for anything approaching high-speed internet access. As such, DSL has been a focus of research and development, leading to significant growth in achievable data rates, from 8MBits Downstream for Asymmetric DSL (ADSL) , through 24 MBits Downstream for ADSL2/2+, and more recently, up to 100MBits symmetrically on short loop lengths (less than 250m) for Very high bit-rate DSL (VDSL).

FTTx and xDSL are both evolving together; as FTTx reduces the last-mile length, xDSL frequency use expands to provide the maximum bandwidth on that short-loop. While in some urban areas, FTTH will become the prevalent form of broadband internet access, xDSL will still be needed for a long time to come for the vast

majority of the populace.

But existing methods of managing the bandwidth allocations for bundles of xDSL lines are either computationally simple, but sub-optimal, or computationally intractable and near-optimal. Much work and research has gone into developing advanced and intelligent Dynamic Spectrum Management (DSM) algorithms, but little attention has been applied to the practical implications of these algorithms and the real-world effect that different forms of that implementation make to performance and tractability.

Massively parallel computing, in the form of GPGPU, is becoming a popular paradigm in computational research, with classically linear algorithms being distributed across these Single Instruction Multiple Data (SIMD) devices giving general speed-ups in the range of 5-500x. The most popular and developed form of GPGPU architecture is NVidia's Compute Unified Device Architecture (CUDA), which leverages the streaming multiprocessor (SM) and high speed memories of their GPU's (usually used for gaming) for use as scientific computation devices.

In this report, the viability of GPGPU accelerated DSM algorithms is investigated,

1.1 Project Specification

School of Electrical and Electronic Engineering and Computer Science
FINAL YEAR PROJECT 2010/2011

Title:	High-Speed bit-loading algorithms for Dynamic Spectrum Management in ADSL
Student:	Andrew Bolster
Supervisor:	Prof A Marshall
Moderator:	Dr J McAllister
Area:	Digital Comms
Research Cluster:	Digital Comms

Digital subscriber lines need to employ bit-loading techniques to improve their throughput; this is known as "Dynamic Spectrum Management" (DSM). Currently all implementations of DSM use level 1 whereby individual lines use a "water-filling" algorithm to adjust the power allocated to each tone in the spectrum, and a central management station adjusts the water-filling parameters for each line in the subscriber bundle. However this approach is sub-optimal whenever the capacity of the overall bundle is considered, hence more recent research has proposed (level 2) more intelligent methods to allocate the power for each bit in each tone, considering both near and far end crosstalk in the bundle. A major problem with the level 2 techniques is their computational complexity which currently renders them practically infeasible, e.g. ISB typically takes more than 1 week to compute the tones for a 10-line bundle. Graphic Processing Units (GPUs) represent a new approach to massively parallel floating-point computations. Moreover the Compute Unified Device Architecture (CUDA) developed by NVidia,represents a framework whereby new highly parallel algorithms can be developed. The main aim of this project is to apply this approach to level 2 DSM algorithms, many of which are highly parallel in their operation.

The objectives of the project are:

1. Become familiar with DSM techniques for Digital Subscriber Lines.
2. Become familiar with the CUDA environment for GPU's and identify a suitable platform.
3. Investigate efficient implementations of level 2 DSM.
4. Develop an implementation of a level 2 bit-loading algorithm using GPU's.

M.Eng. Extensions

1. Analyse the performance of your implementation in terms of speed, cost, and scalability (number of lines).
2. Compare your design with existing implementations.

Learning Outcomes

1. Understand how to use CUDA to programme GPU's.
2. Be able to design bit-loading algorithms for DMT.
3. How to analyse the performance of an implementation.

1.2 Overview and Objectives

The project specification is quite expansive, covering a vast range of areas that I have not encountered directly in my studies but have tangentially met externally. Parallel computation is an active area of research, especially Massively Parallel GPGPU systems such as this. Additionally, there are almost as many DSM algorithms as there are DSM researchers, so algorithm selection and implementation must be very carefully planned and executed to keep within a workable time-frame.

As with any project of this scale, the necessary first stage is a thorough investigation of the problem. Not only the areas of DSM directly relevant to the Level 2 algorithms under analysis, but also the underlying DSL technology to enable generation and implementation of a computational model on which to test the algorithms. This will necessitate the selection of a programming environment that must be held consistent throughout the project, as well as the additional implementation of Optimal Spectrum Balancing (OSB) for comparative analysis. In the interests of not reinventing the wheel, research will also be undertaken to assess the usefulness of any existing DSL models / DSM implementations in terms of development guidance.

Beyond the problem itself, technologies involved in the suggested solution will have to be expansively researched, such as CUDA, GPGPU, and Computational Parallelism, including any recent relevant scholarly works that could be useful.

To summarise, the main points that need investigation will be:

- Problem Research
  - How DSL works
  - DMT communications
  - Dynamic Spectrum Management
  - DSM Levels
  - Existing/previous DSL simulation systems
  - Existing/previous DSM implementations
- Solution Research
  - Relevant Programming Environments
  - Software Profiling
  - Massively Parallel Computing
  - Parallel Computing Architectures
  - CUDA software considerations
  - CUDA hardware considerations
- Development
  - Design and implement DSL system model simulator
  - Implement Optimal Spectrum Balancing (OSB) algorithm
  - Implement Greedy bit-loading algorithm (MIPB) algorithm
  - Implement Iterative Spectrum Balancing (ISB) algorithm
  - Evaluate system performance through the use of profilers and workload visualisation
  - Develop single-device GPU versions of above algorithms
  - Evaluate performance of these algorithms, with respect to CPU-bound versions
  - If viable, implement multi-GPU versions of above algorithms

- Analysis
  - Compare and contrast performance concerns and resource usage of differing implementations
  - Evaluate system cost concerns, and potential future avenues of development.

Chapter 2

Research & Background

The modern Broadband telecommunications network has augmented the already massive effect on society of The Internet, allowing access to information from all over the world to one's doorstep. Broadband roll-out and specifically xDSL has spurred the growth of VoIP technologies, Video chat and streaming services, and always-on connectivity, fundamentally changing how society interacts; in 2010, over 31 million adults in the UK made purchases online, that's 62% of the adult population buying from Amazon, Tesco Direct, and eBay instead of going to brick-and-mortar retailers. [2]

Almost 30% of the UK have access to 'Broadband' networks; defined as having a downstream bandwidth of 144Mbits, triple the number from just five years previously. Nielsen reports that in 2010, over 82% of the population had some form of internet access available to them; this number up from just under 60% over the same period.

In terms of speed, variability is rife within the British Isles. Ofcom, in its 2010 year-end report [3] finds that while advertised and actual ADSL speeds are up 5% between May and December 2010, the disparity between advertised 'up to' speeds does not accurately reflect real-world performance. It can be assumed that this disparity can at least be partially explained by service providers using the theoretical capacity of a DSL network, but that network itself cannot adapt to local conditions. The major component of the growth observed in the modern 'to the home' internet is, in Ofcom's opinion, the adoption of new high-speed local-loop technologies, delivering speeds up to and in excess of 45 Mbps in urban areas.

While this is due to a variety of technologies, xDSL plays a massive role; as FTTx technologies increase the available back-haul bandwidth, xDSL technologies are still needed for the ever-shortening (and ever faster) local subscriber loop. Indeed, it is forecast that by 2012, only 4% of Broadband connections with be complete FTTx lines from subscriber to provider. DSL is still alive and well, and this paper will show that there is still plenty of capacity in existing lines that, if used efficiently, can keep up with subscriber demand for at least the next five years.

2.1 DSL

Digital Subscriber Line networks were originally part of the 1984 ISDN specification. But like most communications technologies, it draws its basis from Claud Shannon's 1948 work at Bell Labs.[4]. DSL operates by overlaying wideband digitally modulated data signals on top of existing baseband voice signals on the POTS; over the phone line. This overlaid signal does not effect the voice service, as DSL uses frequencies from around 4kHz to as high as 4MHz; well above the 300-3400Hzoccupied by baseband voice, so both operations can be one asynchronously and simultaneously with little to no interference from each other[1]

2.1.1 DSL Modulation and Signal Transmission

Orthogonal Frequency-Division Multiplexing (OFDM), or as it is standardised within DSL[6], Discrete Multi-Tone (DMT), is the most common modulation scheme in xDSL, where by a large number of orthogonal (non-interfering) sub-channels, tightly packed in the frequency domain, are used to carry data. Across these numerous sub-channels, data is effectively carried in parallel, and streaming data is encoded and split up across these sub-channels at a relatively low symbol rate[7]. In the case of xDSL, this is generally in the range of 2-15 bits per sub-channel[5].

OFDM is used in a wide variety of wideband communications systems, including 802.11a/g/n, WiMax, and DVB-T/H as well as xDSL, and provides many advantages to service providers[8];

1. Tolerant of Multi-path effects such as fading, and Inter-Symbol Interference (ISI)
2. Tolerant of narrow-band co-channel interference.
3. Comparatively high spectral efficiency against spread spectrum modulation systems.
4. Efficiently implemented using Fast Fourier Transform (FFT)
5. Run-time adaptable to severe channel conditions without advanced equalisation techniques.

But it is to be noted that OFDM suffers from some disadvantages;

1. Doppler shift sensitivity
2. Highly sensitive to synchronisation issues
3. Poor power efficiency due to a High Peak to average power ratio (PAPR) necessitating linear transmission circuitry

The ITU G.992 specifications, or G.DMT are the ITU specified sub-schemas of OFDM that are used in Asynchronous DSL deployments (ADSL, ADSL2, and ADSL2+), and ITU G.993 specifies the newer Very-high-bit-rate DSL technologies (VSDL, VSDL2).

[1] the POTS service can affect the performance of a DSL service if the phone lines are not low-pass filtered and split, but this is now common practise[5]

Other xDSL systems such as HDSL, IDSL, RADSL, SDSL, and SHDSL use a combination of Carrier-less Amplitude Phase (CAP) and Two Binary One Quaternary (2B1Q) modulation schemes, which are not the subject of this report, but are interesting to investigate none the less.

Carrier-less Amplitude Phase modulation was the de facto scheme for ADSL up until around 1996 [9], and was almost completely abandoned for ADSL after the 1999 ratification of ADSL ITU G.992.1 Interoperability standard, but lives on in Rate Adaptive DSL (RADSL), and some HDSL and SDSL deployments.

Two Binary One Quaternary modulation, as the name indicates, uses four signalling levels to represent a two bit, Gray coded input signal, meaning that if a signal was misread by one-level, only one bit error would occur. In xDSL, 2B1Q is used in ISDN over DSL (IDSL), and some flavours of HDSL and SDSL.

As stated, the DMT system as implemented in most A/VDSL systems, operates across hundreds of sub-channels, or slices of the frequency spectrum. How the streaming input data is multiplexed, quantised, and modulated across these sub-channels is largely variable. This Spectrum Management problem will be dealt with in more detail later in this report, but for now, it is suffice to say that data can be dynamically spread across and away from 'poor', noisy sub-channels, and also can be concentrated on 'good', clean areas of the spectrum. This technique is called bit-loading, and entails the use of bit-variable modulation, usually based around variable constellation Quadrature Amplitude Modulation [10], and is demonstrated in the two diagrams in figure 2.

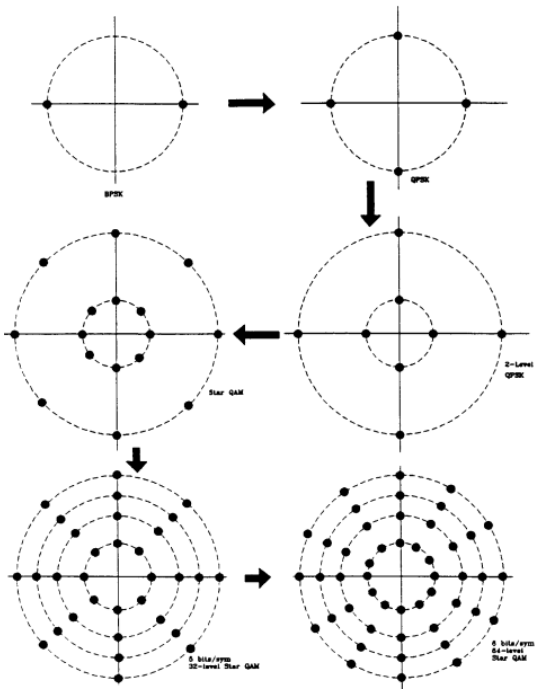


Figure 2: Some QAM constellations used in the variable level scheme

The capacity of a single DMT line is well established, coming again from Shannon's seminal work [4]. His 'ideal' solution, 'Water-filling', simply assigns bit-loads based on the SNR inverse of the line; pouring energy into sub-channels with the best SNR, i.e. most capable of carrying the signal. As the 'best' sub-channels are filled, the adjoining, less capable channels are filled, and so on until no more energy can be added to the channel.

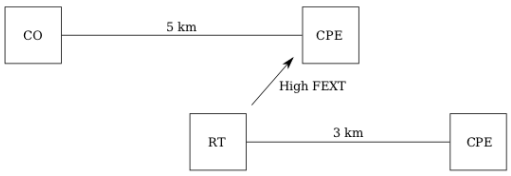
This is an ideal approach, and presents the maximum theoretical capacity of the line. Practical implementations of this, or any other bit-loading algorithm, must sacrifice 'optimality' for reliability. As such, they must implement what is called a 'Shannon Gap' or 'SNR Gap', a sub-ideal limit on the power loaded on a channel to maintain reliable communications. This subject is revisited in section 2.1.2.2, but for the time being, any M-QAM sub-channel constellation, the number of bits that can be encoded within is given in equation 2.1 [4].

$$b = log_2(M) \tag{2.1}$$

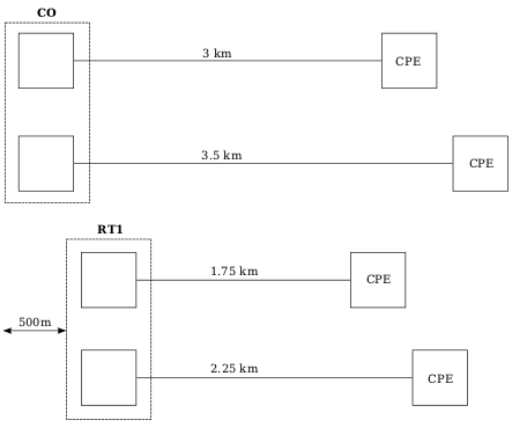
But before going too deeply into the 'problem', first an exploration of the landscape within which the problem lies.

2.1.2 DSL System Architecture

A classical DSL local loop architecture involves a Central Office (CO), with many Line Termination (LT) modems, each connected to a Network Termination (NT) modem at the Customer Premises (CP). Each LT/NT pair has a separate line between them, and for local loops, these lines are wrapped together in bundles of up to fifty individual lines. This bundle then runs from the CO to the CPs, with intermediate CP lines 'peeling off' the bundle. Additionally, along the length of the bundle, additional lines can 'peel in' to the bundle. Some of the wide variety of possible situations are shown diagrammatically in figures 3a,3b.



(a) Possible two line configuration



(b) Possible four line configuration

These bundles make for a highly noisy RF environment; with signals from each line interfering with every other line in the bundle through electromagnetic induction of leaked signal power. This phenomenon is termed cross-talk and is the major limiting factor on the achievable data rates of DSL roll-outs, with no current practical solution.

Crosstalk reduces the SNR on DMT sub-channels on a 'victim' line, reducing the amount of bits-per-symbol that can be transmitted on that sub-channel at a pre-defined constant error rate. The level of SNR reduction is often in the 10-20dB range and as such, is the most dominant noise source experienced within the bundle. [11]

This phenomenon comes in two major forms;

- Near-End Crosstalk (NEXT): signal leakage from LT-LT or NT-NT
- Far-End Crosstalk (FEXT): signal leakage from LT-NT or LT-NT

In the vast majority of current DSL roll-outs, NEXT is effectively eliminated through the use of Frequency Division Duplexing(FDD); i.e the upstream data (NT to LT) is sent on a very different frequency than the downstream (LT to NT), and as such does not cause direct interference with a 'neighbouring' termination point.

The amount of cross-talk (XT) is generally time-constant (excluding severe temperature and humidity changes) and as such can be computationally 'accommodated for' a-priori, but direct vectorisation (we'll find out about that later) and even in-direct cross talk elimination is computationally difficult, if not intractable.

Before one can understand the effects of XT on a DSL system, and especially before one can experiment with different cross talk 'avoidance' algorithms, a realistic dynamic simulation of a functional DSL system must be generated.

2.1.2.1 DSL System Modelling

Modelling a DSL system, instead of opting for 'real-world' values, allows experimentation and observation of channel behaviours that may not be immediately clear from end-point experimentation [2]

Simulation in this case involves the generation of a cross-talk gain (XTG) matrix for an N-line network for each of its K operating channels. This XTG matrix will then be used by the management algorithms to assess the 'cost' or indeed viability of a particular bit-load on a particular line on a particular channel.

As such it is important to understand the electromagnetic transmission characteristics of twisted pair phone lines. This area is largely concerned with the generation of per-line-per-channel transfer functions for interacting sections within a bundle. It is shown in [12] that for standard (Category 3) lines, the following RLCG characterisation is stable up to the 30MHz area, at which point this 'simplified' characterisation veers away from real-world performance due to high-frequency envelope shearing. Category 5 lines are stable with this kind of characterisation up to around 150MHz.

RLCG Characterisation is derived from the per-unit-length two-port model shown in figure 3, which can be viewed as a infinitesimally small section of a segment of transmission line. The RCLG parameters represent Resistance, Inductance, Capacitance and Conductance per unit length of line. The direct gain channel models used in this report are based on the long line approximations taken from [13], which in turn were taken from [12].

This assumption is that (for a long DSL line) the input line impedance V_1 matches the characteristic impedance V_2. Given that assumption as well as;

[2] not to mention that real-world experimentation is time-consuming, expensive, and often incomparable to other results due to a multitude of measurement standards

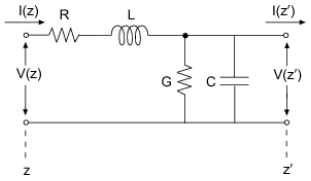


Figure 3: RCLG Characterisation of a two port linear network

- d :line length
- Z :Impedance per unit length:$R \cdot L$
- Y :Admittance per unit length: $G \cdot C$
- Z_l :load impedance
- Z_s :source impedance
- Z_0 :characteristic impedance:$\sqrt{\frac{Z}{Y}}$
- γ :propagation constant:$\sqrt{Z \cdot Y}$

The transfer function for a given DSL line is;

$$H = \frac{Z_0 \cdot \text{sech}(\gamma d)}{Z_s \cdot [\frac{Z_0}{Z_1} + \tanh(\gamma d)] + Z_0 \cdot [1 + \frac{Z_0}{Z_1} \cdot \tanh(\gamma d)]} \tag{2.2}$$

To ensure that this modelled transfer function is smooth with respect to frequency [3], the RLCG values are parametrised thus;

$$R(f) = \frac{1}{\frac{1}{\sqrt[4]{r_{0c}^4 + a_c \cdot f^2}} + \frac{1}{\sqrt[4]{r_{0s}^4 + a_s \cdot f^2}}} \tag{2.3}$$

Where r_{0x} is the DC resistance of copper (c) and steel (s) and a_x are skin effect related constants

$$L(f) = \frac{l_0 + l_\infty (\frac{f}{f_m})^b}{1 + (\frac{f}{f_b})^b} \tag{2.4}$$

Where l_x are low (0) and high (∞) frequency inductance and b and f_m define the transition from low to high frequencies.

$$C(f) = c_\infty + c_0 \cdot f^{-c_e} \tag{2.5}$$

Where C_x represent contact (∞) and other $(0, e)$ capacitances, chosen from measurements.

$$G(f) = g_0 \cdot f^{+g_e} \tag{2.6}$$

[3] necessary due to the large margins of error in characterising 'real-world' lines

Where G_x represent $(0, e)$ conductances, chosen from measurements.

Within this project, as in [13], AWG 24 line parameters were used, shown in table 4.

$r_{0c} = 174.56\Omega/\text{km}$	$r_{0s} = \infty\Omega/\text{km}$	$a_c = 0.0531$	$a_s = 0.0$
$l_0 = 617.25\mu H/\text{km}$	$l_\infty = 478.97\mu H/\text{km}$	$b = 1.1529$	$f_m = 553.76\text{kHz}$
$c_\infty = 50\mu F/\text{km}$	$c_0 = 0.0\mu F/\text{km}$	$c_e = 0.0$	
$g_0 = 234.874fS/\text{km}$	$g_e = 1.38$		

Figure 4: ETSI Parameters used for AWG24 coaxial lines

Using these parameters, figure 5 shows the gain-curve for a variety of lengths of AWG24 cabling produced from equation (2.2)

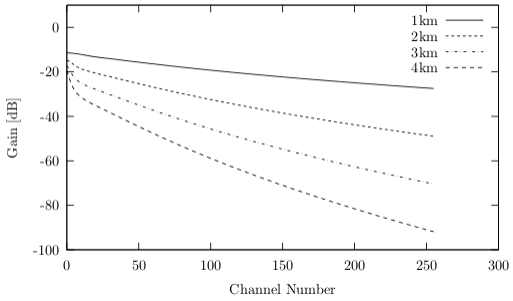


Figure 5: Gain versus channel model for a range of transmission lengths

2.1.2.2 Crosstalk Modelling

The above model allows the generation of 'idealised gains' for a given line, ignoring all external effects, such as the aforementioned cross-talk. To bring cross-talk into the picture, assuming that NEXT is eliminated completely by FDD, we can model FEXT based on the industry standard ETSI 1% [14] model (equation (2.7))

$$|H_{FEXT}(f)|^2 = N^{0.06} K_{FEXT} f^2 |H_{channel}(f, L)|^2 \tag{2.7}$$

The "1%" in this case represents the worst-case-scenario, and in a real deployment would not be encountered 99% of the time. This is partially driven from systemic pessimism, but partially comes from the fact that this worst-case scenario allows for a modelling function that is smooth with frequency.

In most cases, this 1% model is over zealous in its estimations of cross-talk coupling, and also ignores spatiality within the bundle [4]. Using real data from a four sub-bundle, 100x100 DSL binder, the existing model can be modified to more closely match the coupling data of the 'real' bundle.

$$|H^{j,k}_{FEXT}(f)|^2 = |H^{j,k}_{FEXT}(f)|^2 \times 10^{\frac{X_{dB}}{10}} \tag{2.8}$$

Where the modifier X is selected based on a Beta probability distribution of sample data, for example, [15].

[4] i.e., lines in the core of the bundle receive much more cross-talk than those on the skin of the bundle

As stated, DSL is a cross-talk-limited system, and as such, the effect of cross-talkers also effects some of the fundamental assumptions that must be made about the transmission characteristics of the lines in the bundle.

From [16], the approximate Shannon Gap for a DSL system based on M-QAM is derived thus. From the Central Limit Theorem, as the number of cross-talkers increases, the effects of those cross-talkers approximates to a Gaussian Distribution [17]. Therefore, a DSL channel can be approximated as an Additive White Gaussian Noise (AWGN) channel, which is described [4]:

$$C(k) = log_2(1 + SNR(k)) \tag{2.9}$$

This represents the maximum possible capacity (in bits) on a given channel ($C(k)$), ignoring practical considerations such as coding complexity and processing delays. From [4] and [7], the probability of a symbol error for an uncoded M-QAM constellation [5] on a given sub-channel is shown in equation (2.10)

$$P_e \approx N_e Q\left(\sqrt{\frac{3}{M-1}SNR}\right) \tag{2.10}$$

Where $Q(x)$ is the probability of unitary Gaussian variable will will exceed x, given in equation (2.11). In general, DSL systems aim for this probability of symbol error (P_e) to be around $1e^{-7}$.

$$Q(x) = \int_x^\infty \frac{1}{\sqrt{2\pi}} e^{-u^2/2} du \tag{2.11}$$

(2.11) is generally rewritten in terms of the standard error function $erf(x)$, shown in equations (2.12) and (2.13).

$$Q(x) = \frac{1}{2}\left(1 - erf(\frac{x}{\sqrt{2}})\right) \tag{2.12}$$

$$Q^{-1}(x) = \sqrt{2} erf^{-1}(1 - 2x) \tag{2.13}$$

Rearranging (2.10) for M gives (2.14)

$$M = 1 + \frac{SNR}{\frac{1}{3}(Q^{-1}(\frac{P_e}{N_e})^2)} \tag{2.14}$$

Revisiting (2.1), (2.15) can be obtained, expressing the number of bits that can be encoded on a sub-channel.

$$b = log_2\left(1 + \frac{SNR}{\frac{1}{3}(Q^{-1}(\frac{P_e}{N_e})^2)}\right) \tag{2.15}$$

Contrasting (2.15) with (2.9), the uncoded channel gap is defined in equation (2.16).

[5] with an associated number of nearest neighbours, N_e

$$\Gamma_{uncoded} = \frac{1}{3}\left(Q^{-1}(\frac{P_e}{N_e}\right)^2 \tag{2.16}$$

This allows simplification of equation (2.15) with respect to (2.16) into (2.17)

$$b = log_2\left(1 + \frac{SNR}{\Gamma_{uncoded}}\right) \tag{2.17}$$

This characterisation of the Shannon Gap is not quite complete, and represents the best-case scenario within a DSL system, while ignoring potential coding gains from Forward Error Correction such as the Trellis[18] or Reed-Solomon[19] coding schemes.

As such, two additional modifiers are added to the Γ calculation; a performance margin γ_m which allows for SNR 'headroom' to maintain error rates during temporarily bad noise conditions, and a coding gain γ_{eg} which incorporates any error correction modulation in place. This gives a final Γ sum shown in equation (2.18)

$$\Gamma = \Gamma_{uncoded} + \gamma_m + \gamma_{eg} \tag{2.18}$$

In[13] it is demonstrated that cross-talk coupling can be assumed not to have any practically relevant effect on sub-channels other than the sub-channel from which that cross-talk originates, such as when DMT blocks are transmitted and received synchronously or when cyclic prefixing and pulse shaping ("Zipper" DMT[20]) is used.

From[21], equation (2.19) shows the maximally optimal bit-loading on line n of N users, and tone k of K total sub-channels, including the above derivation for FEXT coupling.

$$b_n(k) = log_2\left(1 + \frac{p_n(k)|h_{nn}(k)|^2}{\Gamma\left(\sigma_n^2(k) + \sum_{j\neq n}^{N} p_j(k)|h_{jn}(k)|^2\right)}\right) \tag{2.19}$$

In equation (2.19), the following definitions are provided for clarity;

- $h_{ij}(k)$:The cross-talk gain from user i to user j on channel k
- $p_n(k)$:The power provisioned on channel k for user n
- $\sigma_n^2(k)$:The background noise power experienced by user n on channel k
- Γ:The Shannon Gap derived from (2.18)

By letting $f(b_n(k)) = \Gamma(2^{b_n(k)} - 1)$, (2.19) can be rearranged to (2.20)

$$p_n(k) - f(b_n(k))\sum_{j\neq n}^{N} p_j(k)\frac{|h_{jn}(k)|^2}{|h_{nn}(k)|^2} = f(b_n(k))\frac{\sigma_n^2(k)}{|h_{nn}(k)|^2} \tag{2.20}$$

It is clear that equation (2.20) can be characterised as an N-dimensional linear system of equations, of the form

$$A(k)P(k) = B(k) \tag{2.21}$$

Where

$$A(k)_{ij} = \begin{cases} 1, & \text{for } i = j \\ \frac{-f(b_i(k))|h_{ji}|^2}{|h_{ii}|^2}, & \text{for } i \neq j \end{cases} \tag{2.22}$$

$$P(k) = [p_1(k) \dots p_i(k) \dots p_N(k)]^T \tag{2.23}$$

$$B(k) = \left[\frac{f(b_1(k))\sigma_1^2}{|h_{11}|^2} \dots \frac{f(b_i(k))\sigma_i^2}{|h_{ii}|^2} \dots \frac{f(b_N(k))\sigma_N^2}{|h_{NN}|^2} \right]^T \tag{2.24}$$

As such, the vector $B(k)$, as a function of $b(k$, describes the amount of bits loaded onto each user's line on tone k, and $P(k)$ prescribes the power required on each line to support the vector $B(k)$ of bits loaded onto those lines on channel k. It is the solution and optimisation of this system of equations that is the fundamental limiting factor, and drive for, DSM systems.

The reasoning for this is primarily visible looking at the concept of rate-regions, i.e an N-dimensional surface of possible maximum line bit-loads for each user n. Figure 6 shows the relationship between two given users, R_1 and R_2) within a given bundle. It is evident that one could achieve an very high data-rate on one line, but at great expense to the other. Mathematically, the search for optimal bit-loading is summed up in (2.25).

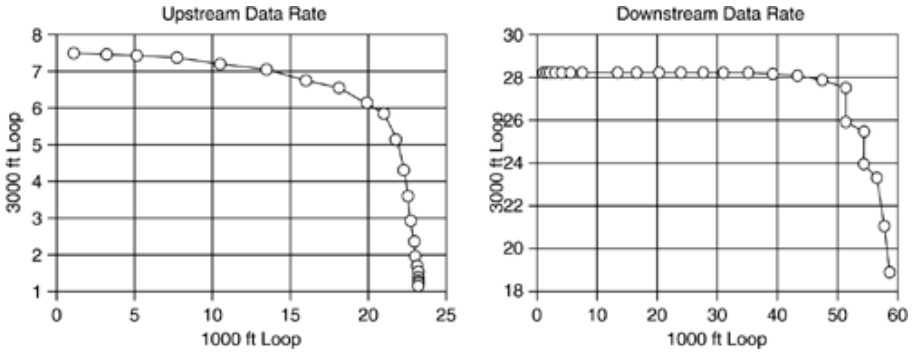


Figure 6: Example of a ADSL rate Region for two users from [5]

$$\max_{\{p_n \in P_n\}_n} \sum_n w_n R_n \tag{2.25}$$

Using an optimal rate-region, of the style demonstrated in equation 2.25 can be generated algorithmically from the above equations. But often these algorithms can be computationally intractable, and less computationally expensive (but sub-optimal) algorithms are used instead, producing reduced rate regions which do not take full advantage of the available spectrum given the same cross-talk and bundle noise characteristics.

In essence, this Spectrum Management Problem, as stated in equation (2.25), can be expanded based on line-weighting; where some lines get 'preferential treatment', either because they are paying more, or as part

of a lead-and-lag rate-compensation system[6]. Thus, the Spectrum Management Problem can be generalised to N users as a maximisation of a weighted sum of data rates within a bundle, as shown in equation (2.26).

$$\sum_{n=1}^{N} w_n \sum_{k=1}^{K} \quad log_2 \left(1 + \frac{p_n(k|h_{nn}(k)|^2}{\Gamma\left(\sigma_n^2(k) + \sum_{m \neq n}^{N} p_m(k)|h_{jm}(k)|^2\right)} \right) \quad s.t. \sum_{k=1}^{K} p_n(k) \leq P_{max,n} \forall n \tag{2.26}$$

The optimisation of this problem in a dynamic near-realtime time-frame is the major focus of DSM research, which is summarised next.

2.1.3 Dynamic Spectrum Management Introduction

A general solution to this rate-region problem has been the use of Static Spectrum Management (SSM), where each line has a pre-defined power spectral density (PSD) profile under which it operates. This is sometimes called Spectral Masking. Unfortunately this masking is done once (or a few times, if lines are added to / removed from the bundle), and additionally assumes that each line will be fully utilised at all times. This a major abstraction from reality, and has driven the development of spectrum management systems that can dynamically re-allocate PSD's based on near-real-time usage. From [22]:

> Suppose that 25 copper lines exist in a cable bundle. Depending on the time (day versus night, weekday versus weekend) and the location (commercial versus residential area), the demand for DSL services varies. For example, the data traffic load at an office is heavy during the daytime and may be close to zero at night. Furthermore, depending on the specific loop topologies, the interference caused by one line to another varies. Some pairs might be physically close in the binder over a very short distance; hence, the interference between these pairs might be very small. On the other hand, other pairs might be proximate along most of the loop length, which leads to a very strong interference between those two pairs . . . Revisiting the example where 25 copper lines exist in a telephone bundle, suppose that in the middle of the night at school, one Ph.D. student is trying to download a large paper from a website using DSL. If static spectrum management is used, the speed is as slow at night as during the day, when many other students are also using their DSLs. However, if the DSL network uses [Dynamic Spectrum Management], the speed at night can be much faster than during the daytime because the bit-rate on the line in use can be optimised to take advantage of the low-noise conditions.

It is clear that with the addition of some usage-based intelligence, the bundle could be much more effectively used. Dynamic Spectrum Management systems allow for power and rate allocations for each line to be dynamically controlled, but are classified into four general levels, based on the amount of coordination between lines, ranging from completely autonomous operation[7] to centralised signal vectoring and control, as summarised in Figure 7, adapted from [23] and [13].

[6] Lag and Lead, terms originally from the field of economics but 'adopted' by the field of control theory, implies a kind of 'lending' system, where by a given variable (in this case the bit-rate/weight on a particular line) can be allowed to 'run rampant' if the system has spare capacity, but that added usage is accounted and is 'paid back' to the system when needed, and often that particular variable will be more 'borrowed from' until its lag/lead accounts are even again

[7] Strictly speaking not DSM but included for completeness

DSM Level	Description	Examples
0	No DSM, Completely Autonomous per-modem Spectrum Management	NA
1	Single line spectrum balancing with politeness and signal impulse control	IWF[a],ASB[b]
2	Multiple line spectrum balancing with spectra controls	OSB[c], ISB[d],SCALE[e]
3	Multiple line signal coordination (vectoring)	Vectored-DMT [24], SVD[f]

[a]Iterative Water Filling
[b]Autonomous Spectrum Balancing
[c]Optimal Spectrum Balancing
[d]Iterative Spectrum Balancing
[e]Successive Convex Approximation for Low ComplExity [25]
[f]Single Vector Decomposition [26]

Figure 7: Summary of DSM Algorithm Level Characteristics with Examples

This coordination (where applicable) is controlled centrally from a Spectrum Maintenance Centre (SMC), which receives data from Customer Premises Equipment (CPE) via an Auto-Configuration Server (ACS), and/or from DSLAMs and LT-side equipment via an Element Management System (EMS). The SMC then collates the received information, and distributes profile settings for the DSLs.

2.1.4 DSM Level 0/1

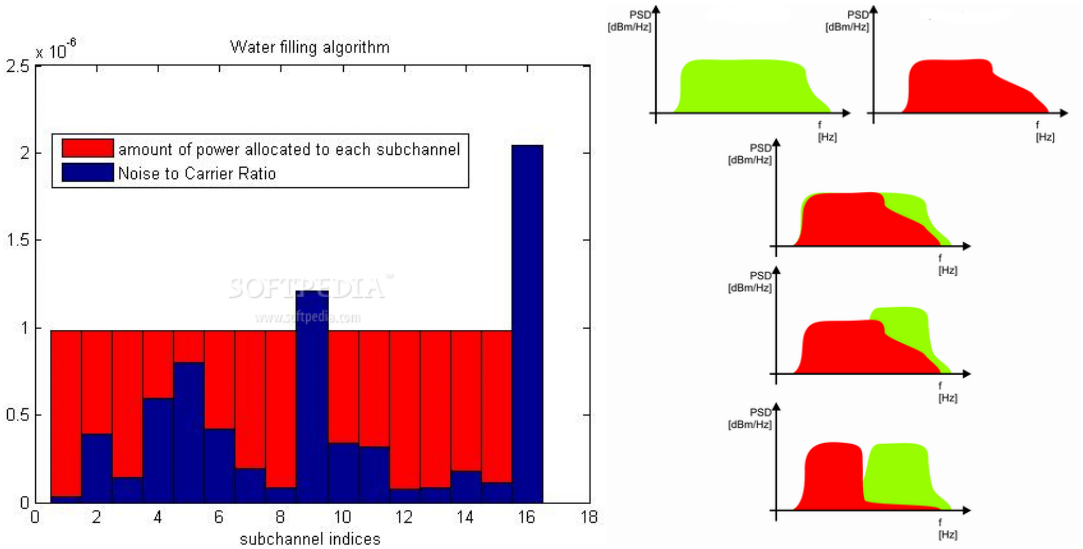
For clarity, levels 0 and 1 are generally grouped, as the only major difference is that while in level 0[8], equipment is completely autonomous in terms of its own spectrum management; level 1 systems receive power and rate limitations from the SMC, within which they must operate. Level 0 is currently the most commen method of DSM utilised currently, but some ISPs are moving to level 1 systems, such as Virgin.

Thinking back to the initial discussion of DMT spectrum balancing, the most common technique for autonomous spectrum assignments is based on Water-filling [4], and refined for DSL in [27] [28]

For the single user case, refer to figure 8a, but in the multi-user case, water-filling attempts must be iteratively refined (Hence, Iterative Water-Filling, or IWF); so as to allow the 'other' lines to see each lines new noise profile, see figure 8b for a graphical example of this. This helps to reduce the effects of the so-called "Near Far Problem", whereby the received signal from cross-talker which is closer to the receiver, is stronger than its direct line transmitter signal.

The water-filling algorithm produces the optimal solution[9], maximising channel capacity for a single user with a continuous bit-assignment [10]. In practise this is incorrect, as integer bits must be assigned to each channel for transmission. When this integer condition is applied, it is known as "Discrete Water-filling". Discrete water-filling has two variations; one which focuses on maximising bit-rate with respect to a stated power budget, known as Rate-Adaptive (RA); and another which focuses on minimising the power requirements

[8]currently the most common method of 'Spectrum Balancing'
[9]It is proven in [29] that for frequencies less than 2MHz, there exists only one Nash Equilibrium point within a distributed DSL network, and although it is possible, no example of multiple optimal rate-region points has been discovered to date [22]
[10]I.e non-integer bit-assignments are permitted



(a) Single User (b) Iterative Multiple User

Figure 8: Example of Water Filling

with respect to a stated bit-rate, known as Fixed-Margin (FM). These two conditions are mathematically phrased in (2.28) (2.27)[12].

$$\begin{aligned} maxR &= \textstyle\sum_{k=1}^{K} b(k) \\ s.t. &\quad \sum_{k=1}^{K} p(k) \leq P_{budget} \end{aligned} \tag{2.27}$$

$$\begin{aligned} min &\quad \sum_{k=1}^{K} p(k) \\ s.t. &\quad \sum_{k=1}^{K} b(k) \geq B_{budget} \end{aligned} \tag{2.28}$$

The resolution of both of these forms of water-filling is surprisingly simple in theory; find the tone with the lowest bit-addition energy [11], and add a bit to it. This continues until either the power or rate budget is reached, depending on the mode applied. This basic algorithm is the Levin-Campello (LC) Algorithm[30].

2.1.4.1 Iterative Water Filling

One of the first forms of DSM, IWF is computationally simple, de-centralised, and relatively easy to implement. The rational of IWF is to limit iteratively perform LC bit-loading and to adapt the power constraints so as to limit the power used in the bundle while maintaining total bundle capacity, thereby lowering the power of cross-talking signals and actually increasing the net bundle data rate greatly when compared to SSM, but is far from optimal, and is not guaranteed to converge, or settle, on on any result. The general algorithm for IWF is shown in figure 9

[11] generally termed $\Delta p(k)$

repeat
  for n=1...N **do**
    Execute LC Algorithm with power budget P_n on line n
    if $R_n > R_n^{target}$ **then**
      $P_n = P_n - \gamma$
    else
      $P_n = P_n + \gamma$
    end if
  end for
until convergence

Figure 9: IWF Algorithm

2.1.5 DSM Level 2

2.1.5.1 Optimal Spectrum Balancing

From [11], OSB optimally solves the Spectrum Management Problem for multiple users, but is highly computationally intensive and generally cannot be practically computed for more than four lines [13]. The avenue taken by OSB in solving the Spectrum Management Problem is a mathematically complex one. The solution is centred around the use of a Lagrangian dual decomposition of the initially states Spectrum Management Problem (2.26). To explain what this process entails, one must look closer at the problem; from [11]

> The fundamental problem is that the total power constraints on the modems couple the optimisation across frequency. As such optimisation must be done jointly across all tones, which leads to an exponential complexity in K. We overcome this problem through the use of the dual decomposition method. This technique allows us to replace the constrained optimisation problem with an unconstrained maximization of a Lagrangian[12] The Lagrangian incorporated the constraints implicitly into the cost function, removing the need for the constraints to be explicitly enforces. As a result, the optimisation can be decoupled across frequency, and an optimal solution can be found in a per-tone fashion. This leads to a linear rather than exponential complexity in K and a computationally tractable problem.

In practical terms, that means that the Spectrum Management Problem can be 'simplified' to include the pan-channel power constraints, meaning that instead of generating many solutions to the general (first line of (2.26)) problem, and then subsequently discarding those as they do not satisfy the bundle power constraints, global power is a focal consideration. The Lagrangian decomposition of (2.26), encompassing the power constraints, is shown in (2.29)

[12] A Lagrangian of a dynamical system is a function that summarises the dynamics of the system, and in the field of optimisation, allows for the joint-optimisation of a dually (in this case) constrained problem through the use of an additional Lagrange multiplier to generate the new Lagrangian decomposition L that implicitly encompasses both (all) constraint functions

$$L = \sum_{n=1}^{N} w_n \sum_{k=1}^{K} log_2 \left(1 + \frac{p_n(k|h_{nn}(k)|^2}{\Gamma \left(\sigma_n^2(k) + \sum\limits_{m \neq n}^{N} p_m(k)|h_{jm}(k)|^2 \right)} \right) - \lambda_n \left(\sum_{k=1}^{K} p_n(k) \right) \tag{2.29}$$

Is was shown by Yu and Lui [31] that this decomposition is an exact match to the originally stated problem for large numbers of channels.

It is also stated in [13] that (2.29) can be tonally decoupled, and a (unfortunately non-convex) Lagrangian expression for each channel generated, as in (2.30), and by optimising for a maximal per-tone Lagrangian, and searching λ space[13], the original problem can be solved.

$$L(k) = \sum_{n=1}^{N} w_n b_n(k) - \sum_{n=1}^{N} \lambda_n p_n(k) \tag{2.30}$$

Since (2.30) is non-convex, an exhaustive search across b_n space is required[14] Even with the Lagrangian decomposition, the computationally explosive nature of this exhaustive bit-field search renders OSB computationally intractable for more than four or five lines. Some run-times from [13] are shown in figure 13

The generalised algorithm for OSB is shown in figure 10

repeat
  repeat
    for $k = 1 \dots K$ **do**
      $\arg\max_{b(k)} L(k) = \sum_{n=1}^{N} w_n b_n(k) - \sum_{n=1}^{N} \lambda_n p_n(k)$
      Solve by N-d exhaustive search
    end for
    $\lambda_n = \lambda_n + \epsilon(\sum_{k=1}^{K} p_n(k) - P_n^{max})$
  until λ convergence
  $w_n = w_n + \epsilon(\sum_{k=1}^{K} b_n(k) - R_n^{target})$
until w convergence

Figure 10: OSB Algorithm

OSB can be augmented using a Branch and Bound searching structure into Branch and Bound OSB (BBOSB). BBOSB is covered in detail in [32], but suffice to say, even with the improved searching structure, BBOSB is still exponential in N, but with a lower complexity coefficient; where OSB is only tractable for 4-5 lines, BBOSB is tractable up to approximately 10 lines [13].

[13] How the λ space is searched is a matter of much debate, which will be met later

[14] For the interested reader, this is a computationally explosive procedure; consider the vector $B(k)$, with each vector index as the bit load on a particular line on channel k, and a maximum bits per tone of 15 (from the DSL standard). For two lines, this represents only 225 possibilities, but for four, its 50625; for 8 its over 2.5 billion, and for a standard 50 line bundle, its a 59 digit number.To put that in perspective, the number of stars in the observable universe is only a 24 digit number. For a given max bits per tone b_{max}, the number of bit combinations for N lines is $b_{max}^N - 1$

2.1.5.2 Iterative Spectrum Balancing

ISB is based on the same Lagrangian decomposition as OSB, but instead of an exhaustive search across the bit-space within the Lagrangian, each line in turn is optimised, i.e a linear search instead of OSB's N-vector-search. It was presented [33] by Cendrillon and Moonen in 2005 and thoroughly investigated by Yu and Lui [31] in 2006, and is near-optimal, but is not guaranteed to converge on a global maximum. [33].

The improvement in computational complexity is great; OSB has a N complexity of $O(b_{\text{max}}^N)$, whereas ISB attains a complexity of $O(N^2)$, meaning that for slightly more practical bundle sizes, ISB is computationally tractable. The ISB algorithm is shown in figure 11

repeat
  for $k = 1 \dots K$ **do**
    repeat
      for $n = 1 \dots N$ **do**
        Fix $b_m(k) \forall m \neq n$
        $\arg\max_{b(k)} L(k) = \sum_{n=1}^{N} w_n b_n(k) - \sum_{n=1}^{N} \lambda_n p_n(k)$
        $\lambda_n = \lambda_n + \epsilon(\sum_{k=1}^{K} p_n(k) - P_n^{max})$
        Solve by 1-d exhaustive search
      end for
    until λ convergence
  end for
  $w_n = w_n + \epsilon(\sum_{k=1}^{K} b_n(k) - R_n^{target})$
until w convergence

Figure 11: ISB Algorithm

2.1.5.3 Multi-User Greedy Spectrum Balancing

It has been shown that while the LC algorithm is optimal in the case of a single line, the straightforward multi-user expansion of this (IWF) is decidedly sub-optimal. Cioffi, Sonalkar, and Lee, in [34] presented a heuristic extension to the Levin-Campello Algorithm, termed Multi-User Greedy Spectrum Balancing. The heuristic applied was, instead of in IWF where each line is individually optimised, the bundle is viewed as a whole, and bits are iteratively assigned to both the line and channel with the minimum cost of addition. This cost of bit-incrementing, which for IWF was simply $\Delta p_m(k)$ where m was the line being balanced, and k was the channel on the line with the lowest additional power requirement; additionally includes the sum of $\Delta p_n(k)$ on all other lines required to accommodate the additional cross-talk generated on the incremented line (2.31).

$$C(m,k) = \sum_{n=1}^{N} \left(\underset{p_n(k)}{b(k)+1} - \underset{p_n(k)}{b(k)} \right) \tag{2.31}$$

2.1.5.4 Multi-User Incremental Power Balancing

In[13], McKinley details the development of MIPB as coming from a critical analysis of Multi-User Greedy Spectrum Balancing. One of the major deficiencies of Greedy is the (significant) possibility of the algorithm getting stuck in local minima, or getting 'deadlocked', whereby bits cannot be added to any lines due to any other line in the bundle attempting to violate its power constraint to accommodate the additional cross talk incurred, (see 2.31). To remedy this, an adaptive power penalty function is used to stop 'clean' lines being continuously loaded up to their power budget, and then 'locking' all other lines, and to instead force all the lines to be more or less equalised as the algorithm progresses. The algorithm for MIPB is shown in figure 12. The final bundle efficiency of MIPB, while non-optimal, is very close, but the real improvement is in runtime performance; 10 lines with rate targeting applied in under five minutes.

repeat
  $\text{argmin}_{n,k} C$
  $b_n k = b_n k + 1$
  for $n = 1 \dots N$ **do**
    $\delta p_{n,k} = \left({}^{b(k)+1}_{p_n(k)} - {}^{b(k)}_{p_n(k)} \right)$
  end for
  for $n = 1 \dots N$ **do**
    for $k = 1 \dots K$ **do**
      $C_{n,k} = \sum_{n=1}^{N} \text{wp}(n) \times \delta p_{n,k}$
      Where wp$n$ is a power penalty function
    end for
  end for
until All tones full

Figure 12: MIPB/Greedy algorithm

		N-User Runtimes (s/sec,m/min,h/hrs)						
Algorithm	Complexity	2	3	4	5	6	7	8
OSB	$O(Kb_{\text{max}}^{N}N^3)$	35.65s	3.24h	*	*	*	*	*
OSB (Cached)	$Kb_{\text{max}}^{N}N^3)$	14.6s	2.12h	*	*	*	*	*
ISB	$O(KN^2N^3)$	32.31s	16.22m	3.93hrs	12.06h	8.69h	28.46h	*
ISB (Cached)	$O(KN^2N^3)$	2.73s	53.57s	9.35m	30.04m	26.74m	1.42h	*
BBOSB	$O(Kb_{\text{max}}^{N}N^3)$	7.12s	15.73m	13.04h	*	*	*	*
BBOSB (Cached)	$O(Kb_{\text{max}}^{N}N^3)$	2.88s	4.2m	3.12h	*	*	*	*
MIPB Bisection	$O(B_{\text{total}}N^2N^3)$	0.26s	9.99s	35.6s	3.71m	12.88m	6.36m	51.72m
MIPB (Cached)	$O(B_{\text{total}}N^2N^3)$	0.19s	3.89s	10.96s	49.97s	*	*	*

Figure 13: Adapted from McKinley '09, 'Caching' implies the use of PSD key-value caching, and * signified unmeasured/invalid execution times

2.1.6 DSM Level 3

Level 3 techniques rely on the concept of signal vectoring, coordinated at a DSLAM, such that all lines are terminated on common equipment. Complete signal vectoring eliminates cross-talk through the generation of a cancellation matrix consisting of block-permuted, diagonal Inverse Discrete Fourier Transform matrices, acting as a Generalised Decision Feedback Equaliser, which, as demonstrated in [35], can eliminate cross-talk, as well ISI in the case of impulse envelope skewing over long lines.

Vectored-DMT (VDMT) can produce optimal bundle efficiencies, but the conditions on VDMT's operation[15] make it inapplicable for commercial use.

Fortunately, DSL Level 3 is not the subject of this project.

2.2 Parallel Computing

Today, most (hopefully all) engineering and computer science students are educated in programming, in one form or another. The current languages of choice;C,C++, Java, and Python, all (classically) conform to what is known as procedural, or imperative, programming, where by instructions are written to be serially fed into a processor, and data flow is dictated by run-time conditional redirection. Ever since the computing innovations of Turning in the 1940's [36] and the later codification of the Von Neumann Architecture (VNA) after his work on EDVAC [16] [37], the concept of a serially operated computers dominated the field of computer science for decades.

Parallel computing, on the other hand, which simultaneously uses many processing units to solve a problem by breaking the problem (or data) set up and distributing this split workload to different processing units[17], has been a niche interest. Parallel computing was for a long time the reserve of a few highly specialised machines created over the years, generally in the fields of medicine, energy or warfare.[18]

The reason for this 'edging out' of parallel computation was simple; Moore's Law [40]. In 1965, Moore, then at the newly formed Fairchild Semiconductor, posited that computational density doubles more-or-less every 18 months. This pattern has held for about four decades. This continuous, explosive, growth drove programmers with computationally intense problems simply to wait 18 months for their applications to run twice as fast on newer hardware, instead of using problem decomposition to solve the current problem in a more distributed way. This period of exponential processing growth in respect to hardware has, in recent years, come up against major blocks; quantum physics[19], the infamous 'power wall'[20], and a general consumer and industrial drive towards low power devices (including 'supercomputers').

[15] Co-location of at least one side of the bundle, computationally expensive pre-computation, inflexibility of applications, requirement for expensive real-time DSP hardware at both bundle terminations

[16] Electronic Discrete Variable Automatic Computer, predominantly used for Ordnance Trajectory calculations and other military applications

[17] Whether instruction pipe-lining should be included in this is a point of debate, which will be dealt with later in this chapter

[18] For example, the Cray-1 was built for fusion plasma dynamics research, the Cytocomputer pipelined processor was built for biomedical image processing in 1980 [38], and the Staran Content-Addressable Parallel computer, built in 1972, was used for Cartographic data processing for military intelligence [39]

[19] Quantum Tunnelling leads to non-deterministic behaviour at around the 16 nanometre mark [41]

[20] If processor speeds were growing today at the same rate as in the mid-90's, CPU's would require a power density equivalent to that of a Saturn V booster stage nozzle [42]

Since the early 2000's, the semiconductor industry has settled on two main paths to stave-off the demise of Moore's Law (in its current form[21]); multi-core and many-core.

Multi-core processing involves the use of a relatively small number of monolithic processing units, very akin to traditional single CPU's, placed on a single die, maintaining the execution speed of existing sequential programs while allowing for direct execution concurrency at lower clock-rates (and hence power) than a similarly scaled single core processor. The latest 'state of the art' in this field is the Intel i7 Sandy Bridge 32nm architecture, with 8 processing cores[43], each of which is a hyper-threaded[22] x86 instruction set processor, giving, in theory, 16 hardware threads of parallelism. The trend in these types of devices has actually almost matched Moore's Law in its simplest form; the number of cores in multi-core chipsets has been roughly doubling every two years.

Many-core computing, on the other hand, involves the use of a relatively large number of 'dumb' cores. While many many-core architectures exist[23] the area in which parallel computing research and applications has been recently most focused is the development and adaptation of consumer graphics hardware to application acceleration and scientific computing, termed General Purpose computing on GPU's (GPGPU).

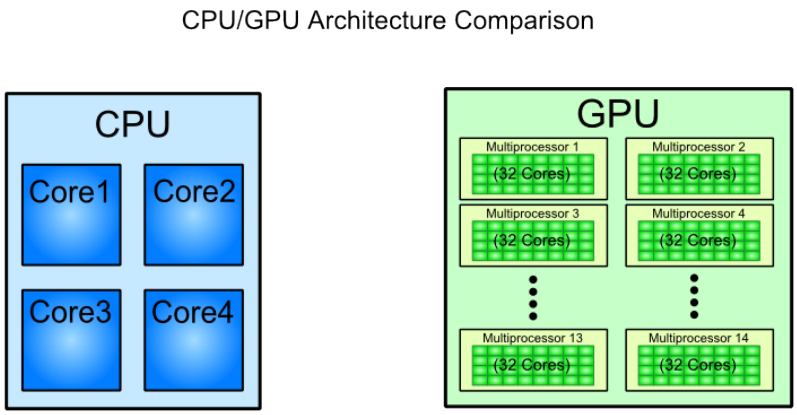


Figure 14: Multicore CPU's and Manycore GPU's have fundamentally different design philosophies

The real difference between multi-core and many-core computing can be seen in Figure 14, whereby a multi-core CPU, such as the Intel Core i5, has a number of large distinct Processing Units (PUs), whereas an NVidia GPU[24] consists of an array of many smaller Streaming Multiprocessors (SM) that themselves contain an array of Streaming Processors (SP)[25], each of which can be considered an individual PU.

While it is not directly relevant to this document, outside of the semiconductor industry, the drive towards massively distributed clusters of not-necessarily-co-located machines [26] has become such a major feature of the scientific computing landscape, that it is currently being used at CERN. Data processing for CERN's LHC project is handled by a worldwide grid of over 200,000 processing cores, with 150 PB of storage, and a theoretical capacity to handle the 27TB of raw data per day coming out of the LHC. This grid system

[21] Moore himself has said 'his' law is the exception to Murphy's Law, in that no-matter what doomsday predictions are made regarding it, The Law seems to perpetuate itself in new forms

[22] Read:two hardware threads per core

[23] IBM's BlueGene/Q architecture, Intel's Larabee Microarchitecture, AMD's FireStream, NVIDIA's Tesla, and Tilera's iMesh to name a few

[24] Used as an example of Many-core computing; the general principals apply across many-core devices

[25] These principals will be covered in more detail in Section 2.2.3

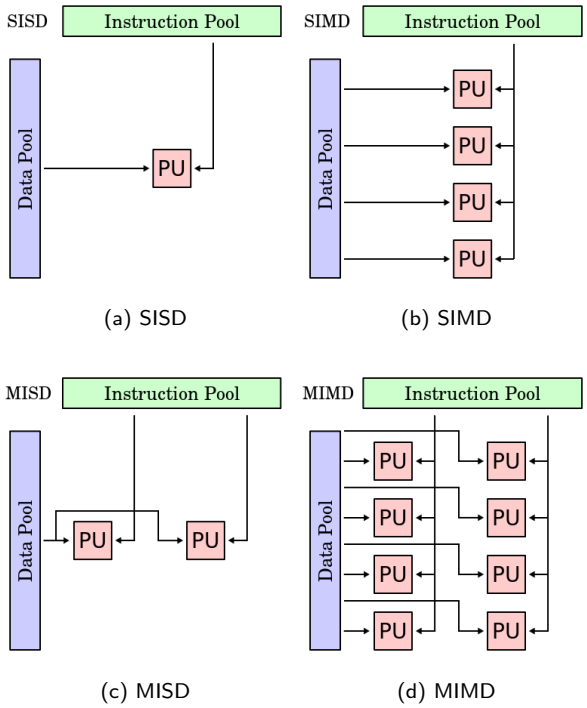
[26] Made famous by SETI@Home in 1999

is inter-connect agnostic, communicating across a mixture of dedicated fibre connections and the public internet backbone. On a smaller scale, programming paradigms such as Message Passing, and Parallel Virtual Machines, allow applications to be executed against an arbitrary number of computing nodes in a local or distributed cluster.

Before looking back at GPGPU and Parallel Programming in detail, it is important to establish the 'families' of parallelism; namely 'Flynn's Taxonomy'

2.2.1 Forms Of Parallelism, aka, Flynn's Taxonomy

Based on the interactions between instructions and data, Flynn [44] classified computation systems in terms of how Processing Units (PU)[27] process instructions and data.



(a) SISD (b) SIMD

(c) MISD (d) MIMD

Figure 15: Flynn's Taxonomy of Parallel Computing Systems

Intuitively, this breakdown is a matrix of data and instruction parallelism. Taking SISD and SIMD separately (as they are the most familiar and relevant), SISD performs a single operation on a single piece of data. It can only process that one piece of data at a time, and can only execute one instruction at a time[28]. SIMD on the other hand takes that one execution instruction, and can operate on many pieces of data, producing many results in parallel. This behaviour is augmented in MIMD, where data and instructions are processed in parallel; this could be within a single device (Such as the Cell Broadband Engine) or distributed across the globe.

Flynn created a very clear and simple map of parallelism. Unfortunately the real world got in the way and

[27] Think of cores or threads of execution
[28] Don't worry dear readers, pipe-lining is coming...

Class	Description	Example
SISD	Single Instruction, Single Data Stream	Traditional Serial Processing
SIMD	Single Instruction, Multiple Data Stream	GPGPU[a]/Many-core
MISD	Multiple Instruction, Single Data Stream	(Rare[b])
MIMD	Multiple Instruction, Multiple Data Stream	Cluster/Grid systems (e.g. LHCGrid)

Figure 16: Flynn's Taxonomy of Parallel Computation

[a]Not really any more, but we're getting to that

[b]This architecture is generally only applied where redundancy is critical, and the 'Multiple Instructions' operate on the signal data stream and must agree. One prominent example is the Space Shuttle Flight Control systems

complicated things. Two major 'complications' will be addressed that have muddied the taxonomic water; SISD Instruction Pipe-lining, and Single Program Multiple Data (SPMD) execution.

2.2.1.1 Pipe-lining

Computer processors are not monolithic boxes that have data pumped into them and the results instantly come out; within even a basic microprocessor, there are a variety of hardware sections that work in sequence, making data and instructions flow through the device to produce the correct result. This sequence is termed the 'pipeline'.

A Generic pipeline consists of four steps;

1. Fetch
2. Decode
3. Execute
4. Write-back

Simply put, Fetch grabs the next instruction to be executed from local memory, Decode processes that instruction, prepares the processor's Arithmetic Logic Unit (ALU) for execution, as well as establishing appropriate data I/O memory channels. During Execute, the calculation is performed, and during Write-back, the result of the calculation is stored back in memory. For the sake of simplicity it can be assumed that each of these stages takes one clock cycle. These tasks are performed by different areas of the chip. This opens up the possibility of having multiple instruction pipelines 'running' at once;

Looking back to Flynn's Taxonomy, it appears at first glance that what was previously a solid SISD processor could be considered MIMD; the data fetched can be different for each pipelined instruction, and those instructions do not have to be the same either. Whether this counts as true parallelism or not is not as important as its performance improvement; where previously one instruction took four clock cycles to execute, with this simple four stage pipeline, four instructions can be processed in 8 clock cycles.[29]

[29]Complications to this system arise if the instructions processed are conditional, but as pipe-lining is not the focus of this project, refer to [45] for more information

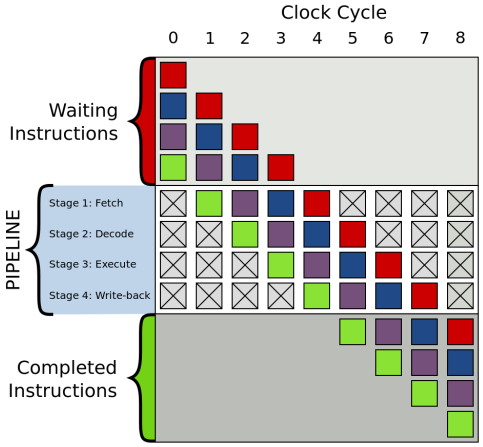


Figure 17: Example of a 4 Stage Pipeline system

2.2.1.2 Single Program Multiple Data

SISD implementations, at least as developed when Flynn stated his Taxonomy, meant that each machine-level instruction was executed in parallel in lock-step. This meant that data-dependent conditional execution simply wasn't possible without suspending the operation of PUs for which that condition was not satisfied. SPMD is a subset of MIMD whereby each PU can continue on at its own execution pace, regardless of the state of other PUs. This is exemplified in the Message Passing Interface (MPI) model, where by a (usually) identical program is executed by all PUs, and to control collective program flow and distribution of data, messages are send and received asynchronously between PUs. These PUs could be processor cores within a single machine, or indeed on the same, multi-threading, core, or they could be on machines on opposite sides of the globe. To add more confusion, in the strictest sense, GPU execution (at least under CUDA) is another different form of parallelism; Single Instruction Multiple Thread (SIMT), which can be though of as a blend of SPMD and SIMD, whereby execution is done is collections of threads simultaneously, and within these collections, instructions are carried out in lock-step as in SIMD, but these collections can be swapped in and out from execution for a variety of reasons, and reorganised based on run-time decisions and resource requirements. The subtleties of this will be discussed in Section 2.2.4.

2.2.2 Principles of Parallel Programming

The concept of splitting a problem set up for separate execution is intuitively simple but in practise very difficult, and has been the major stumbling block in terms of widespread adoption of the parallel programming paradigm. But the potential performance gains afforded from even rudimentary parallelism can be astounding when applied to real world problems.

Examples of problems that are inherently parallel can be found all over the natural and mathematical world; Climate systems can be modelled as atmospheric cells that respond to changes in their surroundings, as can Computational Fluid Dynamics (CFD) and particle physics; Image processing, such as real time video compression can enjoy massive gains as sectors of a video stream can be isolated and processed independently; Linear Algebra, and anywhere where linear algebra is applied[30] is extremely parallelisable in higher-order

[30] It's everywhere

matrices.

2.2.2.1 Gustafson and Amdahl's laws

Performance improvements from parallelism are not perfect however, as any problem will still retain some operations that are inherently serial.

This limitation has been codified by Gene Amdahl in the law that takes his name[46], which states that this serial portion of the problem, however small, will limit the overall speed-up provided from parallising the problem. This is stated mathematically in equation (2.32), where α represents the fraction of the program that cannot be parallelised, and graphically in Figure 18. It is colloquially phrased as:

> When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule; The bearing of a child takes nine months, no matter how many woman are assigned

$$S = \frac{1}{\alpha} \tag{2.32}$$

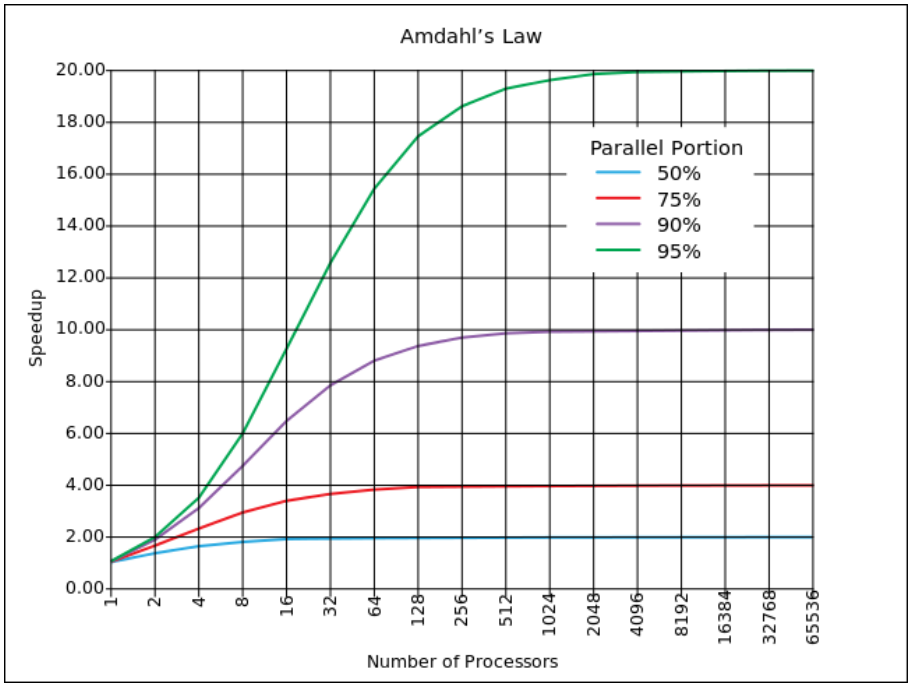


Figure 18: Graph demonstrating the maximum speed-up attained with a variety of parallelisable program ratios under Amdahl's Law

Amdahl's Law assumes that the execution time of the non-parallelisable section of the problem is independent of the number of processors available to the system, and that the problem size is fixed. Gustafson's Law[47] at least partially contradicts Amdahl's Law, by stating that for problems with large datasets, parallelisation is still a worthwhile pursuit, regardless of the sequential portion; effectively implying that while parallelism can't make such problems 'faster', the size of their tractable problem sets can be increased if the serial section does

not 'slow down' with the problem set size increasing. This is stated mathematically in (2.33) , where P is the number of processing units available, and graphically in Figure 19.

$$S(P) = P - \alpha(P - 1) \tag{2.33}$$

Figure 19: Graph demonstrating the maximum speed-up attained with a variety of parallelisable program ratios under Gustafson's Law

The both of these Laws are purely theoretical, and do not include aspects of computation such as communication time between PUs, Memory constraints leading to cache contention[31] or simple processing over-heads involved in running multiple PUs. As such, these Laws can only act as an upper limit to the possible performance of Parallel systems.

2.2.3 General Purpose computing on Graphics Processing Units

Graphics Processing Units were specialised co-processors designed for real-time image processing and generation, with a focus on the fast growing video game industry. The general architecture of GPU's was designed to perform massive numbers of floating point calculations on each video frame, simulating lighting, texturing, and collision detection events for display devices. This led to quite unique design practices in terms of memory management and execution structures. To put this in perspective, main memory access bandwidth from a High End CPU currently stands at approximately 50GB/s[43], a third of the bandwidth of a similar-generation GPU[48]. The idea being that graphics textures are being read many many times over by the collection of PUs, and so needs to be fast.

Around the late 1990's, as this type of hardware became very common on even private desktop machines, the scientific computing community began to use these devices for accelerating highly complex simulations and problems. Up until 2007, in order to accomplish this, the scientific computing problem would have to be 'rephrased' into a graphical problem, utilising a graphics API[32] such as DirectX or OpenGL. This meant that problems such as matrix multiplication had to be rephrased as overlays of transparent textures, as a contrived example. Taking larger more abstract computational problems and decomposing them into graphical operations was far from simple, and was a major block for many institutions with a desire to use these highly parallel devices.

In 2007, NVIDIA released a new reference architecture for its high-end graphics cards, specifically aimed at the scientific and application-acceleration communities; the Compute Unified Device Architecture (CUDA)[49]. CUDA was not just a new C/C++/FORTRAN API, exposing low-level execution and memory control to scientific computing, but was a complete re-write of all intermediate software layers, and included the addition of specific interface hardware, along side the standard graphics interface, dedicated for CUDA computing[50], see Figure 20. In 2007, one GPU chip-set supported CUDA; the G80. In 2008, The Tokyo Institute of Technology's TSUBAME supercomputer became the first GPU accelerated machine in the Top500 World

[31] Whereby working data is partitioned across PUs, and in order to maintain the integrity of these caches with respect to the global data state, additional processing is required

[32] Application Programming Interface: DirectX and OpenGL allowed game developers to have a hardware-agnostic interface to leverage GPUs and other hardware

Supercomputer rankings[33]. By 2011, over 20 different chipsets encompassing over 40 individual manufacturer cards and hundreds of after-market cards, constituting over 100 million CUDA-enabled cards[52] sold across the globe.

In 2008, Apple Inc, in collaboration with NVidia, Intel, IBM and AMD released a C/C++ based framework for mixed CPU/GPU/Manycore/FPGA heterogeneous computing called OpenCL (Open Computing Language). OpenCL contains much more programming abstraction away from the hardware compared to pure-CUDA, and as such cannot be as highly optimised. Even still, NVidia rolled out agnostic device interfaces to OpenCL. The current CUDA architecture is shown in Figure 20

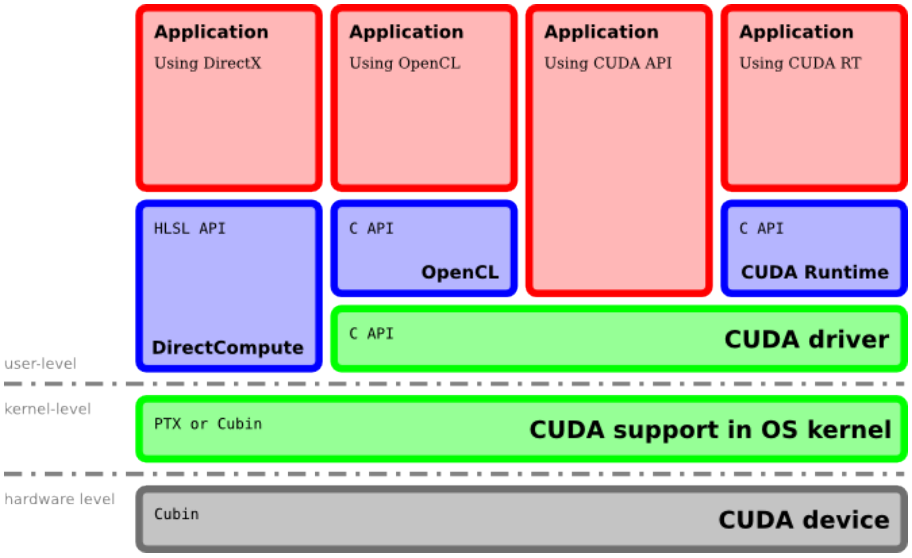


Figure 20: Diagram showing levels of abstraction between Hardware and various APIs

2.2.4 CUDA Execution architecture

A CUDA execution consists of both host (CPU) and device (GPU) phases. The device phases, called kernels, are written in C/C++[34]. Since these kernels reside only on the device, access to main host memory is impossible, and data sets to be worked on, as well as areas of memory to store results, must be set-up by the host on the device before invocation. The amount of parallelism used by the kernel is decided per-kernel invocation, but the kernels themselves must be written with this level of parallelism in mind; there are no magic tricks in CUDA. To understand this, the low level architecture of the GPU must be investigated.

Starting from the top down, a host machine can have multiple GPU devices, which can all be individually addressed for asynchronous execution in parallel. Below this level, and as shown in Figure 21, there are logical 'Grids', which contain logical 'Blocks' of threads. These Grids and Blocks are the fundamental form of execution parallelism. As shown in Figure 21, Grids can be thought of as two dimensional arrays of Blocks, and Blocks are thought of as three dimensional arrays of Threads. It is these threads that actually execute any particular workload.

As stated, the level of parallelism is defined at the kernel invocation stage and (until very recently[35]) only

[33] TSUBAME made it to 29th in the world rankings, with the addition of 170 Tesla S1070 systems[51]
[34] As we'll see, wrappers for other languages exist, but there is always an intermediate C/C++ stage
[35] The latest Fermi devices support multiple parallel kernel invocations, under which SMs are assigned different kernels based

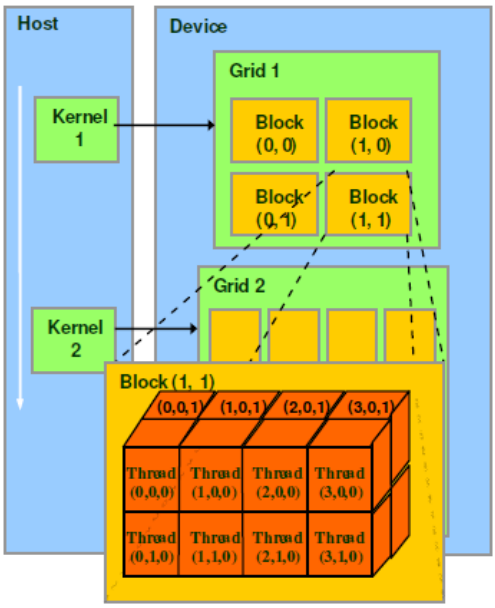


Figure 21: Diagram showing levels of execution between host and CUDA device operations

```
1    //Setup dimensions of grids and blocks
2    dim3 blocksPerGrid(65535,1,1);
3    dim3 threadsPerBlock(64,8,1);
4
5    //Invoke Kernel
6    kernelfunction<<<blocksPerGrid,threadsPerBlock>>>(*functionarguments);
```

Figure 22: Example CUDA host code segment, showing kernel invocation

one kernel can run on a single device at a time. Following the SIMD model, parallelism is attained by per-thread self-indexing. In the case of CUDA, each thread could generate a unique 1D execution index using a combination of runtime variables that are pro grammatically exposed through the CUDA Driver API, as shown in Figure 23. In this particular example, it assumed that both Grid and Block dimensions are 1D. This is useful in this case for scalar multiplication of linear arrays, and could process input data containing $2^{16} \times 2^{10} = 2^{27}$, or over 67 million values[36]. CUDA introduces several additional keywords to the C language, in this case "__global__", which indicates that the function is executed on the device, but can be called from the host. A summary of these function declaration keywords is shown in Table 2.1

```
1    __global__ void multArray(float *array, float multiplier){
2      int 1Dindex = blockIdx.x*blockDim.x+threadIdx.x;
3      array[1Dindex]=array[1Dindex]*multiplier;
4    }
```

Figure 23: Example CUDA kernel, showing 1D index identification

on a proprietary load-balancing algorithm
[36]For the latest generation of Tesla GPUs

Function Declaration	Executed by	Callable From
__device__ void someKernel	device	device
__global__ void someKernel	device	host
__host__ void someKernel	host	host

Table 2.1: Table of CUDA Function Declaration Keywords and their use

For more parallelism, and more context relevance, consider scalar multiplication of large matrices. The previously stated indexing scheme could be used sequentially, i.e taking each row of the matrix in turn and farming the computation of that row to the GPU, but as stated, CUDA allows (actually encourages) multi-dimensional indexing, so each thread execution could be tasked with modifying multiplying one matrix element by 2D addressing, as shown in Figure 24. This form of parallelism theoretically allows for up to $2^{16} * 2^{16} * 2^{10} * 2^{10} = 2^{52}$ or about 4.5 quadrillion threads, (i.e operating on a square matrix of side 2^{27})[37].

```
1  __global__ void multMatrix(float *matrix, float multiplier){
2    int x_index = blockIdx.x*blockDim.x+threadIdx.x*;
3    int y_index = blockIdx.y*blockDim.y+threadIdx.y*;
4    matrix[x_index][y_index]=matrix[x_index][y_index]*multiplier;
5  }
```

Figure 24: Example CUDA kernel, showing 2D index identification

Moving into the practical realm, once a kernel is launched, the CUDA runtime system generates the corresponding logical grid of threads. These threads are assigned to execution resources such as shared memories and thread registers[38] on a block-by-block basis[39]. These resources are organised into streaming multiprocessors (SMs), the number of which vary depending on the particular hardware, but usually around 15 are active. These SMs can each be assigned up to 32 thread-blocks, each of which is executed on a separate Streaming Processor (SP) core. These cores can handle up to 48 threads in parallel. In the case of the Tesla C2050, this means that over 20,000 threads can be 'simultaneously' executed.

Note that this does not limit the grid and block dimensions; groups of threads, termed warps, are swapped in and out of the SM's regularly, making execution tolerant of long-latency operations such as global memory access. This warping of threads is also an important concept for thread-divergence; when runtime-dependant conditional statements in kernel execution have different branching behaviours in threads that are in the same 'warp', the warp is actually executed twice; once for the major condition, and once for the minor condition. Between these two executions, the results obtained from the 'minor' path are discarded and during the minor execution, the results from the major path are also discarded. It is for this reason that conditional behaviour should be avoided in a GPU environment (Figure 25 demonstrates this. Adapted from [53]).

As mentioned, resource allocation is partially decided upon the memory requirements of a particular kernel. These and other memory related concepts are covered in the following section.

[37] Due to Memory and Hardware considerations, this is a ridiculously contrived metric
[38] These will be covered in detail in Section 26
[39] For the sake of simplicity, the following section discusses the hardware capabilities of the latest generation of Fermi cards, specifically the Tesla C2050 Workstation card

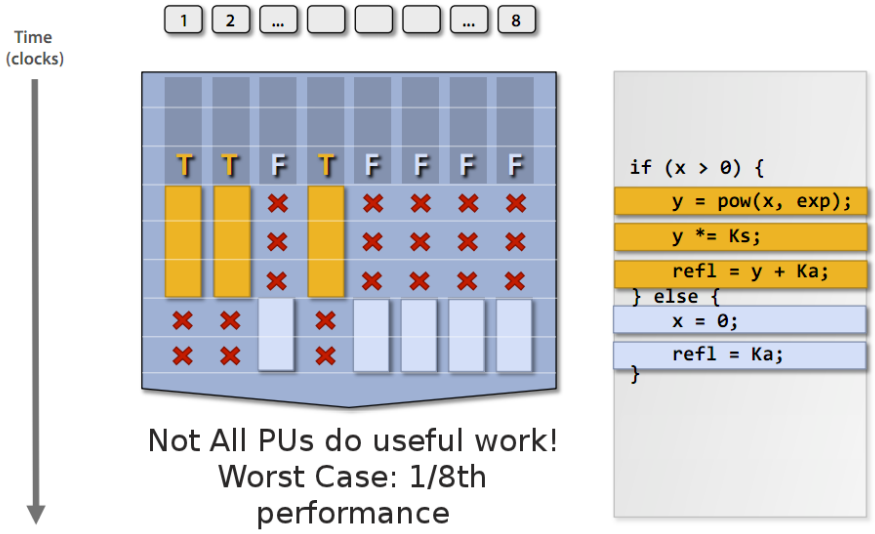


Figure 25: Diagram showing thread divergence operation under CUDA, adapted from SIGGRAPH 2008 proceedings

2.2.5 CUDA Memory architecture

Utilising the levels of parallelism demonstrated previously, one would expect massive performance improvements to be a given. This is not the case, and 'lazily parallelised' applications generally achieve only a small fraction of the potential speed of the underlying hardware, and most of the time, the limiting factor is memory latency. To understand why this is the case, it is important to investigate the CUDA memory architecture.

CUDA devices have three major levels of memory; Thread local, Block Shared, or Grid Global. This architecture is displayed diagrammatically in Figure 26, and summarised in Table 2.2.

Memory	Scope	Lifetime	R/W	Usage	Speed
Register	Thread	Kernel	R/W	Automatic variables other than arrays	Very Fast
Local	Thread	Kernel	R/W	Automatic Array Variables	Very Fast
Shared	Block	Kernel	R/W	__shared__	Fast
Global	Grid	Application	R/W	Default	Very Slow
Constant	Grid	Application	R	__constant__	Slow

Table 2.2: Table of CUDA Memories and some characteristics

In order to get data to and from the device, Global, Constant and Texture memory is read-write accessible from the host; any other memory allocation is done on a per-thread basis.

Texture memory is a particular area of constantly declared memory that is logically represented as a 2D array, and is augmented with a distributed cache of 2D localised values from last access and as such is significantly faster than Global memory. This behaviour is particularly useful for applications such as linear algebra and CFD.

Due to their scientific ubiquity, parallelisation of linear algebra systems is a heavily researched field, leading to highly customised libraries available, such as BLAS (Basic Linear Algebra Subprograms) and MKL (Math

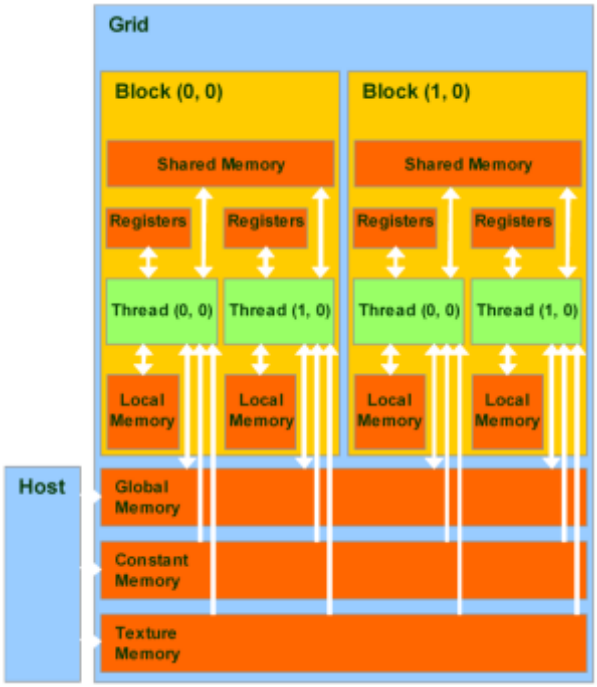


Figure 26: Diagram showing CUDA memory access architecture

Kernel Library). cuBLAS is a CUDA library specifically optimised for most Level 1, 2, and 3 BLAS functions running on GPU, and this library and others like is is heavily used within the research community [54].

This competitive optimisation in all linear algebra systems means that defined linear algebra functions are a natural benchmark for cross-comparison between parallel CPU and GPU implementations; each candidate library doing the best they can do with the hardware. And frankly, GPU wipes the floor with CPU, as in Figure 27.

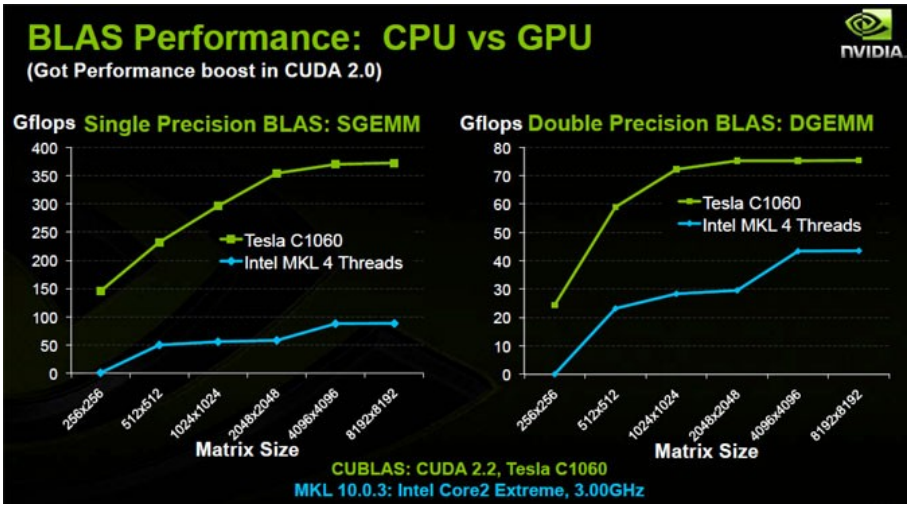


Figure 27: Image Courtesy of NVidia Corp. Showing CPU/GPU comparison of highly optimised BLAS libraries

In [50], a variety of matrix multiplication kernels designed for matrices of many thousands of elements are shown with a variety of optimisations. A Naive implementation is shown, as in Figure 28 that performs at

(only) 17.2 GFLOPS[40]. With a few modifications, a similar kernel (Figure 30) can perform at 47.5 GFLOPS; nearly 280% faster. The major modification is the use of what is called 'shared' memory, i.e memory that is common to a thread-block.

In Figure 28; matrix pointers A, B, C point to the two input and one output matrix respectively, where global memory has been allocated and moved onto the device by the host application, and the kernel is invoked with the width of the matrix to stay in memory bounds. Each block of threads will calculate a section of the output matrix, as shown in Figure 29.

```
__global__ void matmulNaive(float *A, float *B, float *C, int WIDTH){
  Tx = threadIdx.x; Ty = threadIdx.y;
  Bx = blockIdx.x; By = blockIdx.y;

  X = Bx * blockDim.x + Tx;
  Y = By * blockDim.y + Ty;

  idxA = Y * WIDTH;
  idxB = X;
  idxC = Y * WIDTH + X;

  Csub = 0.0;

  for (i=0; I < WIDTH; i++) {
    Csub += A[idxA] * B[idxB];
    idxA += 1;
    idxB += WIDTH;
  }

  C[idxC] = Csub;
}
```

Figure 28: Example CUDA kernel, showing Naive parallel matrix multiplication with Global memory access

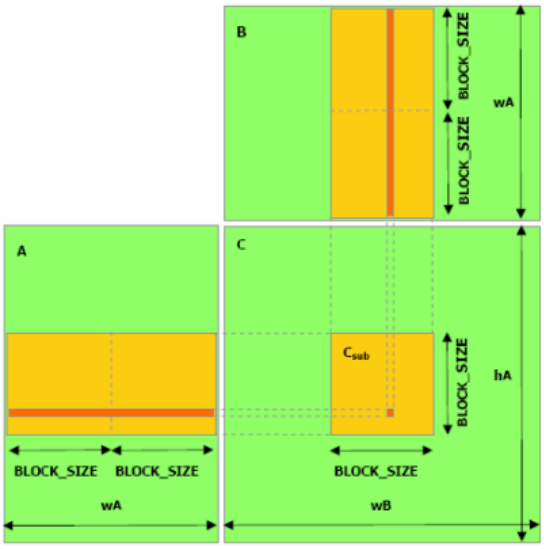


Figure 29: Naive matrix calculation with memory access by a single block highlighted

[40] Giga-Floating Point Operations per Second

In this example, every access to A, B, or C (lines 15 and 20) is from/to Global memory. This access is orders of magnitude slower than the access to thread-local variables such as the A and B indexes. One improvement that can be made initially is to use block-shared memory. This is demonstrated in Figure 30. In this case, each thread retrieves one element from each input array, and stores it in block-shared memory, i.e the threads collaboratively copy the data required for whole-block execution. Note the __syncthreads() in lines 19 and 22; this CUDA call instructs each thread in the block to wait for all other threads in the block to come to the same execution point before continuing. In this case this is to ensure that all of the relevant elements have been copied by all of the block-warps before trying to do any actual calculations. The operation is similar to previous, as shown in Figure 29, except that the outer while loop makes each thread 'step' across the input matrices. This has the effect of greatly reducing the number of Global memory accesses, and has the added benefit of increasing what is called coalesced memory access.

```
1
2         __shared__ float As[blockDim.x][blockDim.y];
3         __shared__ float Bs[blockDim.y][blockDim.x];
4         Csub = 0.0;
5         Tx = threadIdx.x; Ty = threadIdx.y;
6         Bx = blockIdx.x; By = blockIdx.y;
7
8         X = Bx * blockDim.x + Tx;
9         Y = By * blockDim.y + Ty;
10
11        idxA = Y * WIDTH;
12        idxB = X;
13        idxC = Y * WIDTH + X;
14
15        while (idxA<WIDTH) {  // iterate across tiles
16          As[Ty][Tx] = A[idxA];
17          Bs[Ty][Tx] = B[idxB];
18          idxA += blockDim.x;  idxB += blockDim.y * WIDTH;
19          __syncthreads();
20          for (i=0; i < 16; i++) {
21            Csub += As[Ty][i] * Bs[i][Tx];
22            __syncthreads();
23          }
24        }
25        C[idxC] = Csub
```

Figure 30: Example CUDA kernel, showing Naive parallel matrix multiplication with Shared memory access

Coalesced memory accesses are simply batching of memory reads and writes, where all (or most) threads in a warp access a linearly contiguous data space, i.e the k^{th} thread in a given warp accesses the k^{th} element of an array. This way, the 'block' SP can perform these reads or writes as one instruction, instead of each individual thread accessing individually. In this case, matrix A accesses are largely coalesced, as each thread is grabbing its own element within a row of A, but B accesses are uncoalesced. One solution, that will not be investigated here, is performing an transpose on B and then performing the more complex multiplication; this operation would only be useful for fairly large Matrices.

Looking beyond shared memory and memory coalescing schemes schemes, CUDA exposes many memory access interfaces for handling different arrangements of data, and for more detailed information refer to [55].

In summary, the performance of CUDA, and generally any, parallel application is dependant on many factors;

from efficient runtime resource allocation; memory types and access techniques; and most importantly, input data dimensions. This is demonstrated when matrix multiplication algorithms are used applied to 'smaller' matrices; Figure 31 shows that for small matrices, $N \ 180$, CPU-bound calculation is significantly more performant than GPU-bound solutions. From this section it is clear why this is so; memory latency, kernel processing overheads, and a minimal workload all work against a parallelised GPU implementation. Where the problem set is large, for instance, warp-swapping is used to continue to efficiently provision resources to queued warps while other warps are waiting on memory retrievals. This economy of scale is an important design heuristic when developing for GPU and will significantly influence the allocation of work in this project.

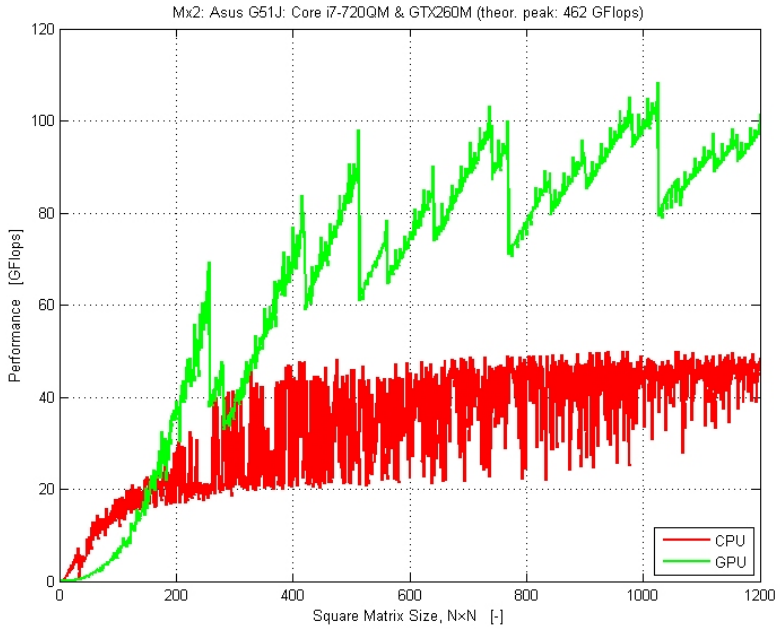


Figure 31: Image Courtesy of AccelerEyes Showing lower order CPU/GPU comparison of MATLAB Matrix Multiplication using the Jacket framework

2.3 Opportunities for Parallel Decomposition of DSM Algorithms

With an understanding of the mathematical challenges presented in the DSM problem, and the current availability of Parallel Computing technologies (specifically those presented by innovations in GPGPU), it is worthwhile to pause and analyse potential decomposition techniques that could be applied to algorithmic acceleration.

In [13], McKinley observes that the majority of computational time accrued in the generation of optimal and near-optimal bit-loading configurations is in the calculation of power spectral densities of candidate bit-loads. This naturally stems from the previously discussed exponential relationship between number of lines, maximum bits-per-tone, and the resultant number of possible bit-combinations. The calculation of PSDs is the solution of N linear systems, as stated in equation (2.21).

As discussed in Section 2.2, parallel computation is ideal, especially under the CUDA model, for solving systems of linear equations with high N. The downside is that to utilise parallelism effectively, the number of

systems must be quite large (100+, see Figure 31). Since in general DSM bundles consist of 50 lines, and range up to 100, there is no rational justification for offloading this work to existing libraries such as cuBLAS, as they will perform significantly worse than equally optimised CPU-bound libraries.

Even so, the small N also can be an advantage; considering the previously discussed memory architecture of CUDA, it is feasible to create a small, customised, linear system solver that resides in-thread; i.e, each thread solving one small system of equations.

To restate the DSM problem in general; the process of generating optimal bit-load configurations is a N dimensional optimisation problem, and as such stands as one of the most difficult problems in computation, which has not completely been 'solved' in either the sequential or parallel realms [56].

This allows a simplified reclassification of the previously discussed DSM algorithms in terms of purely their qualitative searching techniques; OSB exhibits N dimensional exhaustive search behaviour, ISB exhibits behaviour similar to segmented linear search schemes, and MIPB takes a heuristic increment-and-search approach.

The range of these behaviours immediately indicates that no 'one-size-fits-all' solution is going to work; each problem will have to be tackled individually.

2.3.1 Parallel OSB

Even though OSB is a computationally explosive searching solution, it would still be worth while to attempt to parallelise it, as it is the 'simplest' of algorithms, and its core can be explained quite simply; For each channel, calculate the PSD's of every possible bit-loading combination, subsequently find the Lagrangian cost for this bit-load scheme on this channel, and thus return the maximally optimal bit-loading combination for the entire bundle on this channel.

It's clear even from this simple statement that parallelisation will be effective; Each thread can calculate the PSD and subsequent Lagrangian for each bit-combination in parallel, then a reduction can be used to find the maximum Lagrangian and bit-load. This presents a prarallelisation across Channels, Lines, and bit-permutations, and would be relatively simple to leverage multiple devices in this scheme; each device is assigned a channel range or a channel queue, and calculates the local optimisation for the channel, returning its result back to the host.

Intuitively, this should give significant speed-up's for lower numbers of lines (i.e up to 4 or 6), but due to bit-combination-explosion, will remain intractable for higher line counts.

2.3.2 Parallel ISB

ISB segments the search problem of OSB such that instead of searching every possible bit-loading combination for every line, per-user bit-loading is searched on each tone, so that in the lowest level, there are $O(N^2)$ combinations as opposed to OSB's $O(2^N)$. After searching a single line, the global bit-load for that line is updated, and the next line is searched.

From a parallelisation perspective, this global updating of bit-loads is concerning as it creates much more opportunity for memory contingency problems as well as non-optimal SM occupancy where the number of users is relatively small (i.e. below 8), as each line-search will have to wait on the search for the previous

line. That said that larger numbers of lines, the problem should be highly scalable by having thread blocks collaboratively searching individual line loads and continuously updating a global record of bit-loads. Indeed there is the possibility of having larger thread blocks that individually search different lambda values in parallel, simplifying bisection search. Another possibility is to invert the ISB looping constructs such that all channels can take one 'step' in parallel.

As such, one would hope for at least channel and bit permutation parallelism, but due to the incremental update nature of ISB, could not exhibit line parallelism. This represents a much smaller parallelisation problem which will be significantly more difficult to shoehorn into a multi-device structure due to data dependency, but one would predict that the combination of reduced computational complexity of ISB, and this style of parallelism, that GPU ISB will be very very fast compared to even GPU OSB.

2.3.3 Parallel MIPB

Due to the sequentially coupled nature of the MIPB algorithm, it is inherently difficult to parallelise effectively. MIPB only needs to update one N psd vector at each bit-increment, and this could only really be accelerated across N threads. This factor alone indicates that MIPB may not be directly parallelisable in its simple form, but some future adaptations of MIPB utilising the deterministic nature of 'bit-increment chains' presents an algorithmic possibility of pre-computing the bit-increment paths on each channel and using these weighted increment trees to allocate bit-loads based on power constraints.

In any case, MIPB will be implemented as a CPU bound comparison, with an attempt at a GPU implementation, to assess the comparative merits of for instance GPU ISB/OSB against MIPB.

Chapter 3

Development and Solution

After performing research into the problem we now have a better understanding of exactly what needs to be done, and some of the theoretical and implementation based conditions that must be adhered to. It is clear that current implementations of DSM algorithms cannot be put to practical use due to computational intractability with standard sequential hardware, and that a move towards parallelisation has significant potential benefits. But in order to approach the algorithmic problem, first a DSL simulation framework must be created within which to experiment, and this is no small undertaking.

To start with, a stable software base must be selected, that incorporates rapid development due to the expansive scale of this project without sacrificing too much in the way of performance, as well as interoperability with the CUDA API, and is supported by a strong community that can be leveraged to ensure that development is not halted by 'silly' problems.

In[13], McKinley had the same task on his hands, and due to its close-to-the-hardware speed, selected the C programming language. While this was indeed a very fast [1], C can often be prohibitively obtuse, with many arcane design patterns and structures that do not aid in rapid prototyping.

Instead, the Python programming language was selected (in agreement with Dr McKinley) as the base of this project. Python is an interpreted, high-level language, originally created by Guido van Rossum in the 1980's. Two of the biggest draws to Python as a general-purpose language are its flexibility[2] and its extension interface; a significant amount of Python's standard module library is built in C/C++, such that these modules are effectively as fast as straight-C implementations. This is of particular importance with the most popular scientific math library in Python, called Numpy, which is entirely C based and operates very close to the metal, including transparent C-variable assignments and other functionality that preserves the speed of C and the higher level functionality of Python. Additionally, the wealth of Python Profiling[3] tools available means that iterative optimisation (at least of the Python section of the framework) would be painless and fast.

So far, it has been established and justified that Python satisfies the first of the conditions for a stable project base; performance rapid development. To satisfy the second, a Python project called PyCUDA should be

[1] Technically record breaking

[2] Python supports Imperative, Object Oriented, Aspect Oriented and Functional Programming constructs interchangeably

[3] Profiling is the process of recording a software execution and monitoring the performance and runtime of its constituent functions, providing the developer with a 'look under the hood' to establish what areas of code could be optimised to give the maximum performance improvement with minimum wasted time

noted.

PyCUDA is a complete CUDA API wrapper for Python, incorporating advantages such as automatic device selection and initialisation, dynamic memory management, kernel templating and direct variable passing, Just In Time (JIT) kernel compilation and invocation, and run-time access to device characteristics and status for dynamic block and grid dimensioning. Unfortunately PyCUDA is lacking in some areas, particularly debugging support, Python Profiling integration (i.e CUDA code will have to be profiled and analyses separately from any other functionality), and automatic multiple device allocation.

Considering these together, the Pros of Python + PyCUDA greatly outweigh their disadvantages. That said, the largest disadvantage is the lack of truly automatic multi-device provisioning, and this is not a major obstacle. Consider this scenario; A single CUDA kernel execution outside of PyCUDA could execute on multiple devices [4], for the planned problem complexities and data structures, host-to-device memory transactions would eat up any added performance incurred from this style of execution. It is much simpler (and most likely faster) to partition the problem space across the number of available devices, and with the flexibility of Python, this is workable with only a few lines of additional code.

Outside of the programming language itself, utilising a free and open source Distributed Revision Control (DRC) called Mercurial allows for tracking changes and activity in the project, as well as serving as an automatic backup system, through a service called BitBucket, which also provides and issue tracker and on-site wiki for development related notes and tasks.

3.1 Solution Development Task list

In order to fully specify the development of a solution, a task list with appropriate milestones was required to monitor progress and maintain framework stability. This was implemented specifically so that experimental development (or branching) could be explored freely in different development areas (to be detailed), while still allocating time for framework and algorithm re-factoring to leverage shared-object functional decomposition. This task-list mirrors the expanded project objectives in Section 1.2, with additional developmental details.

1. Framework Development and Verification of CPU-bound algorithms
   (a) Implementation of Object Oriented DSL bundle simulation framework, with software hooks for algorithm interfacing and standardised result and instrumentation storage formatting.
   (b) Verify Channel Matrix generation against known dataset
   (c) Pure-Python implementation of OSB
   (d) Verify OSB bit-loading against known dataset
   (e) Pure-Python implementations of MIPB and ISB
   (f) Verify results against known dataset
   (g) Refactor algorithm object structure to reduce functional duplication
2. Development and Verification of GPU-bound algorithms
   (a) Single GPU implementation of OSB
   (b) Verify results against CPU version
   (c) Multi-device development and implementation of OSB

[4] This is technically true only for the latest version of CUDA; at time of writing, 4.0 RC2

(d) Verify results against CPU version
(e) Single GPU implementation of ISB
(f) Verify results against CPU version
(g) Refactor algorithm object structure to reduce functional duplication

3.2 Simulation Framework Architecture

The architecture of the solution will consist of a simulation framework, emulating the gain-characteristics of an arbitrary DSL bundle structure, with software hooks exposed to any algorithms implemented.

The fundamental datum required for DSM algorithms is a matrix of crosstalk gains between all lines. This matrix must be generated incorporating the physical relationship of different lines to each other, as well as simulating the material characteristics of the lines themselves in order to calculate the direct gains (i.e the 'cross talk gain' between line n and itself, or the matrix diagonal).

Once generated, this matrix would describe the bundle of lines, and this inherent value led to the decision to develop the Simulation Framework as an Object Oriented model; the bundle object contains line objects that have individual values, such as their line and network termination locations, noise characteristics, computed SNR, preferential line rates, etc... Additional to these values, each line object exposes internal functions that perform operations on 'itself', such as calculating the far end crosstalk experienced by that line on a particular channel, and the RLCG transfer function over its length.

The bundle object itself defines the system configuration, and is created from a list of line values read from a file; initially it is planned that this file would contain the LN/NT distances of the line, and if required, the desired rate for that line, but the same theory could be applied to allow mixed material bundles, per line noise characteristics, and other pertinent values. The bundle object's main aims are to initially generate the cross-talk matrix, and subsequently act as an abstraction layer between the DSM algorithm and the individual line values; i.e. the bundle keeps internal arrays to store the power and bit ratings for each line for each channel, which can be operated upon by an external algorithm, and subsequently updated by the bundle itself.

The initialisation of the bundle object (i.e. the generation of the cross-talk gain matrix) can be thought of as a triply nested loop of transfer function calculations, and can be summed up in one sentence: On each sub-channel, for each line (the victim) calculate all the inter-line transfer functions between the victim and every line in the bundle (including itself, but that's slightly different).

Calculating the direct gain (i.e. the transfer function between the line and itself), the line object simply returns its own transfer function across its length. However, for inter-line (FEXT only, in this project) transfer function calculation, the situation is more complex since lines don't necessarily have the same run length and location; there are nine possible combinations of over-lay between two given lines (plus the case where they don't share any length, hence no FEXT). These are detailed in [57] and indicated in Figure 32, and this structure was the basis for McKinley's work in this area, but an improved, 'case-less' implementation was generated to segment the line lengths into head length (i.e where one line goes further than the other towards the CO end of the bundle), shared length (where both lines occupy the same sector of the bundle), and tail length(where one line goes further towards the CP end). Each length subsequently has a sector insertion loss (transfer function) and these are multiplied to give the final line transfer function which, incorporating

a NICC[5] FEXT model function, provides the full length gain response between the two lines on a particular channel.

The operation of segmenting these lengths and producing the cumulative transfer function is numerically subtle but conceptually simple, and with the transfer function configured to return 1 for invalid length values (i.e non-positive lengths), the cumulative transfer function product is effectively self selecting which (if any) sector length values to 'ignore' given different theoretical cases.

Mathematically,the generation of the inter-line transfer function proceeds thus;

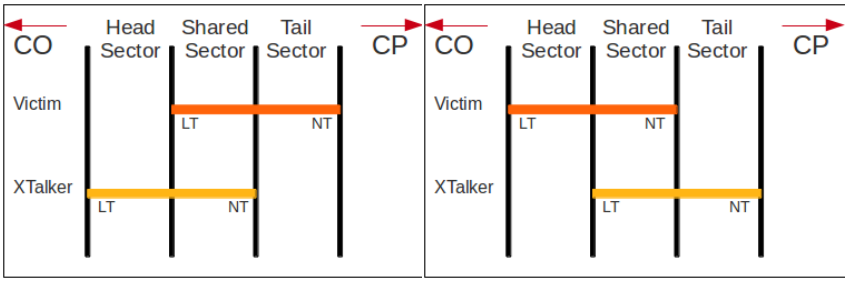
$$\begin{aligned} L_h &= (V_{lt} - X_{lt}) \\ L_s &= \text{abs}(\max(V_{lt}, X_{lt}) - \min(V_{nt}, X_{nt})) \\ L_t &= (V_{nt} - X_{nt}) \end{aligned} \tag{3.1}$$

$$\text{insertion loss}(L, F) = \begin{cases} \text{transfer function}(L, F) & \text{if } L > 0 \\ 1 & \text{if} L \leq 0 \end{cases} \tag{3.2}$$

$$\begin{aligned} H_h &= \text{insertion loss}(L_h, f_k) \\ H_s &= \text{insertion loss}(L_s, f_k) \\ H_t &= \text{insertion loss}(L_t, f_k) \\ H_{\text{total}} &= (H_h \times H_s \times H_t) \end{aligned} \tag{3.3}$$

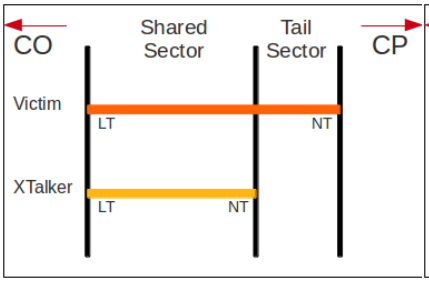
This is explained diagrammatically in Figure 32, and the source code for this function as used is in Appendix A.

[5]NICC is a UK based communications interoperability standards body

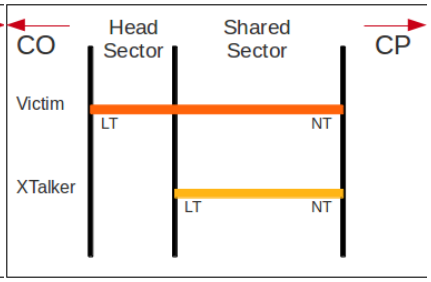


(a) Case 1

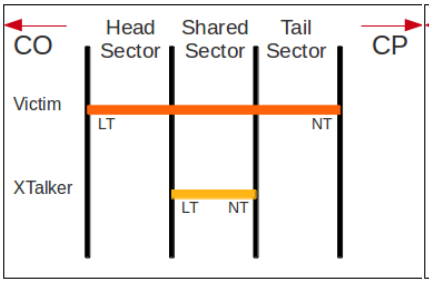
(b) Case 9



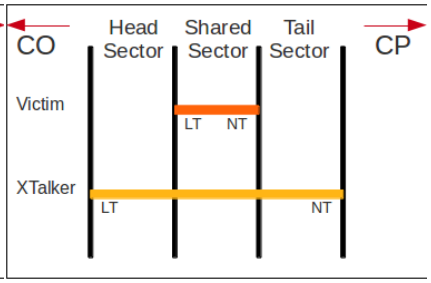
(c) Case 2



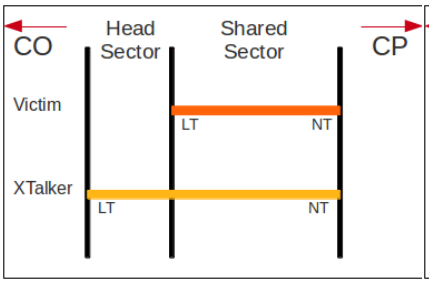
(d) Case 6



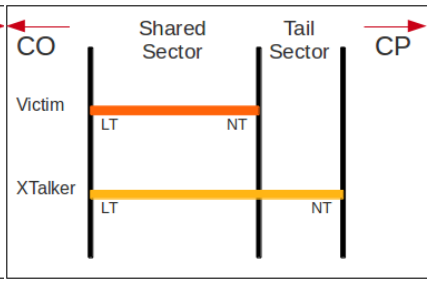
(e) Case 3



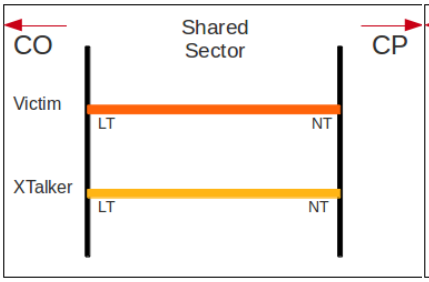
(f) Case 7



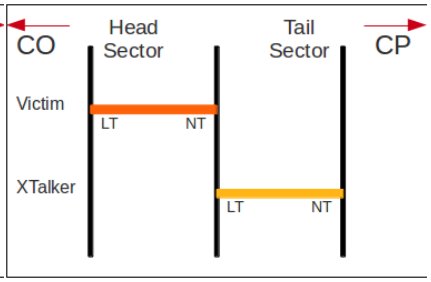
(g) Case 4



(h) Case 8



(i) Case 5



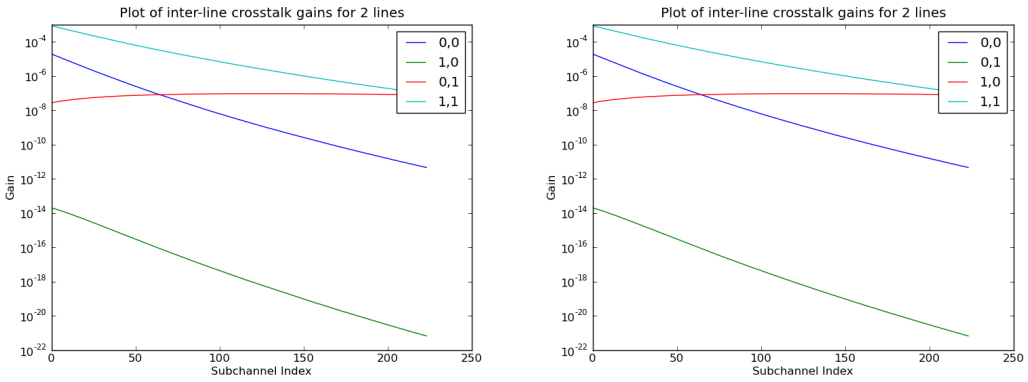
(j) Case 10

Figure 32: Case-based explanation of improved FEXT modelling algorithm. Note, Case numbering rearranged to highlight upstream/downstream FEXT case inversion

As noted previously, incorporation of a Beta probability distribution sample data offsetting can be used to more accurately model the stochastic relationship in inter-line gains based on the locations of those lines in the bundle. This is accomplished by scalar multiplication of the per-channel cross-talk gain matrix with a $N \times N$ sub-matrix of static gains measured from a 'real' bundle (See Equation 2.8). One advantageous side effect of this appears in bit-loading of bundles with some 'identical' lines; due to the numerical instability of some bit-loading algorithms and specifically the generation of line power spectral densities, assigned bit-loads can fluctuate violently between identical lines, producing very impulsive spectra. The relatively tiny gain adjustments applied to different identical lines in the simulated bundle present enough of a 'difference' to overcome this behaviour.

Following from the general Object Oriented architecture, and the bundle object effectively being the fundamental core of the simulation system, this object also maintains control of GPU-based algorithm specific functions, and in the case of multiple GPU operation, maintains persistent thread-pool and task queuing references within a GPU object.

The Initial stage of verification for the system was the comparison of generated Channel Matrices to those found in[13]. Due to the different math libraries utilised between McKinley's implementation and this, direct comparison on numerical results is not reliable as a verification method, but test results obtained match those from McKinley to within IEEE floating point specifications. A better comparison is a visual one (Figure 33).



(a) McKinley

(b) Implementation Results

Figure 33: Visual comparison of known-good channel matrix (McKinley) and Python implementation show that they are identical

3.3 CPU-bound Algorithm Development and Verification

Before tackling the GPU implementation head on, in order to gain a more in-depth and practical understanding of the different DSM algorithms, pure Python, CPU-bound implementations of OSB, ISB and Greedy bit-loading were created. Due to the previously mentioned Object Oriented model, this process was greatly simplified by having each different algorithm class being a 'child' of a generic 'Algorithm' class; inheriting from that parent operations common to all algorithms, such as defining and instantiating universal variables, performing timing operations, verifying PSD 'sanity', file I/O functionality, and general 'build-up/tear-down' operations. This reduced the complexity of each algorithm class (slightly) and ensured that each algorithm

was being tested in a consistent and repeatable fashion.

Since OSB is the 'simplest' DSM algorithm, it was implemented first (without per line rate targeting[6], but with PSD caching), largely from the work covered in [13], with operation as stated previously. Due to Python's efficient syntax and the functionality available in the Numpy math library, this was a fairly straightforward task in terms of implementing mathematical formulae in a structured way, but with commonly neglected areas such as algorithmic convergence and numerous boundary cases that are not covered in any of the technical papers cited, this was a process of iterative 'run it; it breaks; find new edge case; implement edge case; repeat'. Such edge cases range from ensuring that while PSD's can have negative values, that a negative PSD value indicates a 'failed' bit-load[7] to ensuring that assigned line Lagrangian co-factor bisection did not reduce factors to infinitesimal values given very different line sections.

The in this case, the generation of power spectral density values for each attempted bit combination is accomplished using the Numpy linear algebra library. This development and implementation was carried out in parallel with the generation of the general simulation framework. This joint development allowed for very early-stage verification of the work currently applied, specifically verification of cross-talk matrices and subsequent bit-loads against values derived from [13] and [11].

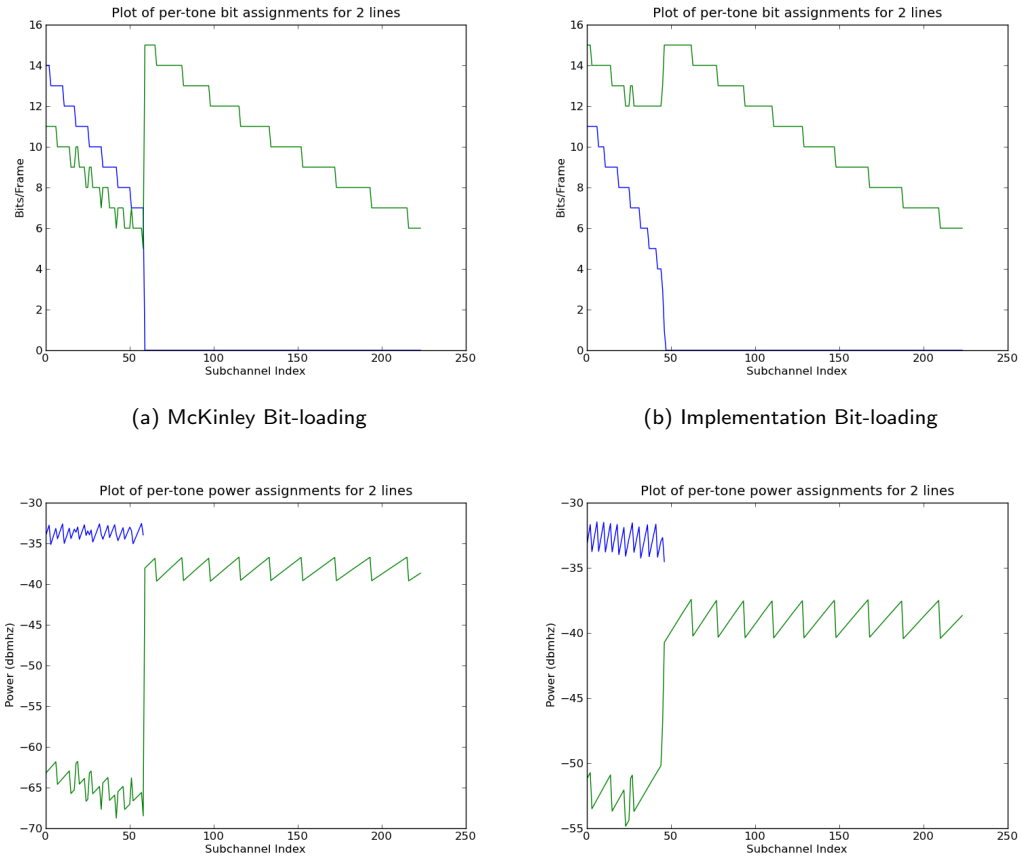
The development of the line and bundle objects, as well as the general framework structure, allowing for programmatic 'hooks' for objects (classes) for different algorithms, automatic generation of post-bit-loading graph data represents more than half of the development time applied to the project[8].

Verification of the OSB algorithm was done in a similar way to the verification of the gain-matrix generation; comparison with known-good results. In this case this takes the form of analysing the resultant PSD and bit assignments for the lines graphically, as shown in Figure 34. While these do not match each other perfectly, this can be explained by two factors;propagated floating point representation differences, and updated beta-offset implementations. The McKinley implementation was built on a 32-bit floating point representation system, while the presented implementation leverages Numpy's support for 64-bit floating point values. Additionally the selected beta-offset model application in the bundle differs from the McKinley implementation in that in the McKinley model, random beta offset values are applied to the lines sequentially where in this implementation, an offset matrix is applied globally to the bundle using Numpy functionality. Given the inherent numerical instability of the calculation of the Lagrangian sum, fractional changes in cross-talk gains can significantly affect the final resultant bit-loading assignments, while maintaining a close rate-approximation between the two implementations.

After OSB was implemented, verified and their operation confirmed by Dr McKinley, Greedy bit-loading (MIPB) with rate targets was developed from [13]. One modification that was made to MIPB as in [13] was the introduction of a modified line weight updating algorithm that used the ratio-to-target rather than distance-to-target as a weight-shifting factor, giving on average half the number of iterations while maintaining expected bit-loads.[9]

An implementation of ISB soon followed, and verified in the same manner. One particular additional version was made of the ISB algorithm in preparation for GPU parallelisation; loop reversal of the inner optimisation step. 'Classic' ISB iterates over each line individually, internally repeating bit incrementing steps until bit-

[6] Rate Targeting is not a focus of this project, but was explored for fun anyway
[7] And propagating this information back to the algorithm root before wasting any more processing time on this attempt
[8] Seeing it condensed into just over a page of text is slightly depressing!
[9] It is for this reason that MIPB results tend to be slightly skewed with respect to the McKinley implementation.



(a) McKinley Bit-loading

(b) Implementation Bit-loading

(c) McKinley PSD

(d) Implementation PSD

Figure 34: Visual comparison of known-good (McKinley) and Python implementations of OSB show that they are near-identical

load convergence is achieved, shown in Figure 35. An alternative but equivalent loop construct is to do the per-channel incrementing inside a global bit-convergence loop, as shown in Figure 36. This manipulation is safe since each channel is ideally independent to the power conditions of other tones.

for all channels **do**
  repeat
    for all lines **do**
      $\text{argmax}_{b_k} L(k)$
      By 1-d exhaustive search
    end for
  until Bit-load Convergence
end for

Figure 35: Standard ISB Loop construct

As part of second-stage verification and re-factoring, significant functional portions of OSB and ISB were moved into the Algorithm super-class, such as lambda bisection and rate metric functions, since both algorithms do largely the same thing within the outer loops of the implementation. Since the only differences

repeat
  for all channels **do**
    for all lines **do**
      argmax$_{b_k} L(k)$
      By 1-d exhaustive search
    end for
  end for
until Bit-load Convergence

Figure 36: Exchanged ISB Loop construct

between the two algorithms is their inner optimisation function, these are included in this document as Appendices B and C.

For completeness, figure 37 and 38 show like for like comparison of bit load and power spectra between this solution and the McKinley implementation given a standard two line near-far scenario (as shown in 3a)[10]

3.4 GPU-bound Algorithm Development

3.4.1 Retrospective analysis of CPU bound applications

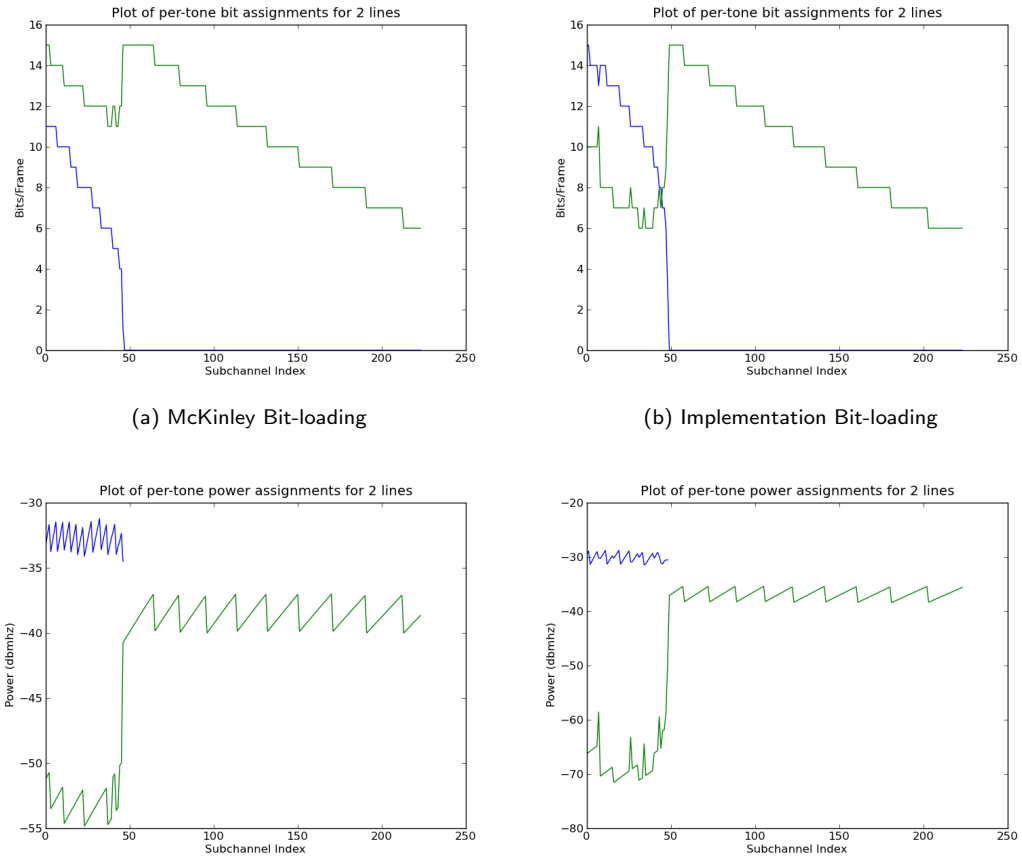
Python's profiling architecture allows for detailed analysis of 'hot-zones' within applications, i.e. the areas within a program where most time and resources are spent over the course of execution. In order to see a 'practical' workload, for the following analysis, a four line near-far bundle was used3b. In order to visualise the profiled-executions, the RunSnakeRun application was used to generate squaremap representations of time spent in different functions; larger areas represent a greater total time spent in that area of computation (Figure 39).

3.4.1.1 Conclusions

From these profiling investigations, it is clear that the largest computational bottleneck is the calculation of power spectral densities for different bit-load combinations. This is exemplified particularly in OSB and ISB; the Lagrangian calculations ('L_k') clearly take up the vast majority of computation time, and in the case of MIPB, the culprit is the recalculation of the Δp values per tone; again, the primary operation is calculation of PSD's. As previously discussed, OSB and ISB, from an algorithmic perspective, lend themselves to parallelisation, but the lack of on-board linear algebra library to calculate individual $N \times N$-system solutions is a serious problem; under-test, using the cuBLAS library to calculate 4-system solutions was actually 4% slower than using the built-in Numpy linear-algebra system on the CPU side[11]. It is clear that for such small matrices, there is no advantage to using standard GPU libraries. And unfortunately, no alternative libraries

[10] Notice that occasionally the low-channel bit-allocations are swapped between the two lines, i.e. line A in McKinley may have the bit-allocation of line B in this implementation below channel 50. Consultations with McKinley state that this behaviour is acceptable.

[11] To try and make this test fair, The calculation was repeated 256 times with pre-generated random inputs on both the GPU and CPU side, with these repeats parallelised on the GPU



(a) McKinley Bit-loading

(b) Implementation Bit-loading

(c) McKinley PSD

(d) Implementation PSD

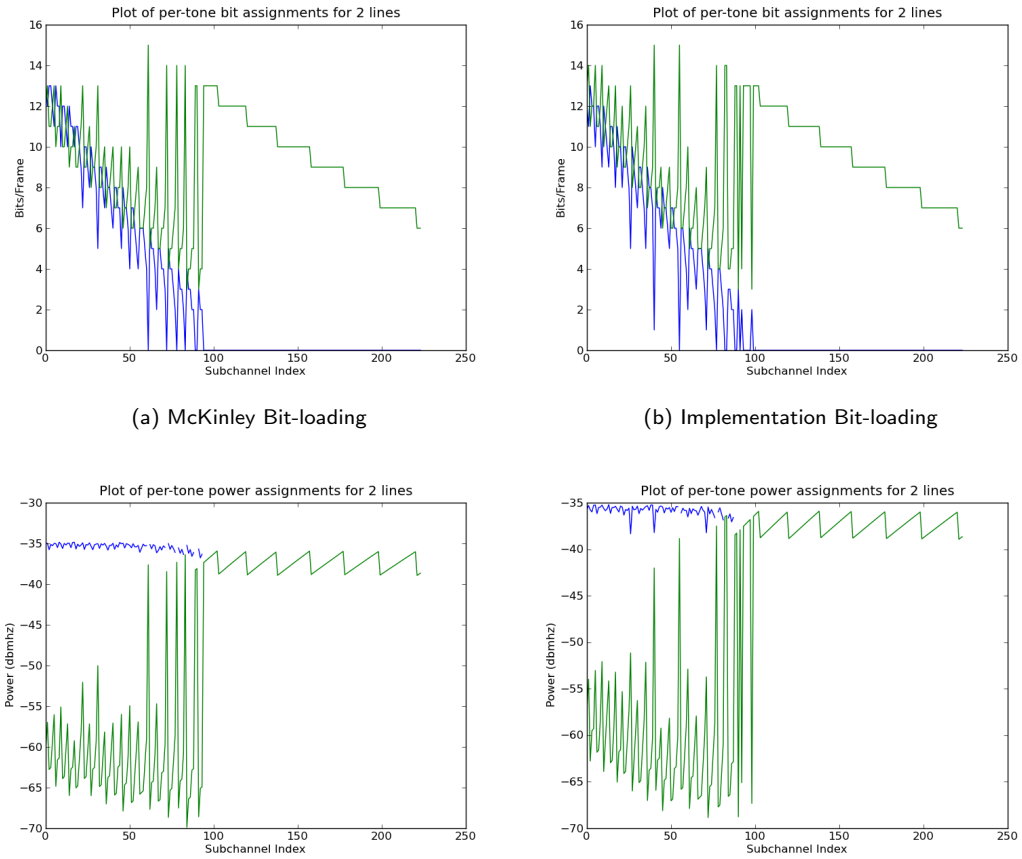
Figure 37: Visual comparison of known-good bit-load and power spectra for Greedy (MIPB), with rates targeted to OSB, show that the implementations are equivalent

could be found, and since the CUDA device cannot access any host data, C implementations such as GSL, LAPACK, LINPACK, or EIGEN could not be used. This leads to the need to create a custom, stripped down, linear system solver[12].

Source code is availiable from https://bitbucket.org/bolster/multiuserdsm for all of the applications (and more) used within this project.

Due to the numerical instability of the systems involved, the linear solver must be able to handle arbitrary matrices, as well as being able to gracefully handle failure without bringing the whole GPU down. This led to the selection of a customised maximally pivoted LU Decomposition algorithm[13], pieced together from [58], [59], [56] and [60], and the final solution arrived at is shown in Appendix D. This algorithm does not leverage CUDA parallelism. The reason for this is two fold; a natural parallelisation scheme for this algorithm would be for a collection of threads to collaboratively work on a single matrix, but this arrangement would not efficiently occupy the GPU's processing units until the number of lines being tested was greater than at least 8. A Secondary reason is that from the perspective of OSB, having each block calculate a single bit-permutation would greatly limit the number of permutations able to be tested simultaneously, and as such the same kernel

[12] This was quite possibly the most technically difficult and frustrating section of the entire project
[13] A Blog post discussing the development of this function is availiable at http://bit.ly/eH3WH6



(a) McKinley Bit-loading

(b) Implementation Bit-loading

(c) McKinley PSD

(d) Implementation PSD

Figure 38: Visual comparison of known-good bit-load and power spectra for ISB, show that the implementations are equivalent

that could optimally handle up to 2^{16} permutations (for example, four lines) in one execution would have to be executed eight times to compute the same result. Further, due to the conditional nature of LU Decomposition, utilising per-thread solving on locally related datasets reduces the total amount of warp divergence, increasing overall speed. Experimentally, using block-shared matrix solving was approximately 23% faster than thread-solving for single executions, but when put into a realistic cradle of input values and ranges, was 15% slower. This was largely due to the need for repeated executions, as well as the previously stated warp-divergence issues.

The profile results also show that the calculation of optimise_p is the section of the OSB and ISB algorithms most heavily in need of optimisation. To note in this section is the difference in how the "asarray.numeric" numpy functions within optimise_p are executed between OSB and ISB; these functions are involved in the PSD caching operation; every time a PSD value is requested, the function arguments are hashed, and a dictionary of past values is searched for that hash. This operation greatly reduces the number of linear algebra operations performed in OSB especially (attaining a cache hit ratio of over 98%, meaning that the algorithm only needs to execute 2% of the time that it is actually called)[14], but does so at the expense of

[14]ISB has a lower, but still significant, cache hit ratio of just over 80%, but since ISB executes approximately 5% the number of PSD calculations as OSB, this caching is less significant. MIPB by comparison has a hit ratio of 0%; no PSD is asked for twice, so caching is largely useless unless using rate targeting, and even then hit ratios average between 10-20%.

system memory (for a six line network, this cache easily exceeds 8GB). If the calculation of the PSD's can be sufficiently accelerated, this cache could be done away with completely, greatly reducing the total memory footprint of the system, and therefore the cost of execution.

3.4.2 OSB: Avenues of parallelisation and problem decomposition schemes

OSB power optimisation is a naturally parallel algorithm; calculating all possible permutations for all users for all channels, finding the Lagrangian for the best combination on each channel, and loading it. This multiple-level parallelism makes it perfect for GPU execution, but also for multiple GPU execution; each GPU can be assigned blocks of channels or using a channel queue, while each device computes all the permutations for that channel. Power Optimisation in OSB has three distinct sections; Generation of A and B matrices from individual bit-load permutations, PSD calculation for each permutation, and Lagrangian calculation for that PSD using assigned lambda and weight values. This leads to the possibility of using three independent kernels, with persistent memories across executions (i.e. no need to move memory around during a single optimisation).

This logical decomposition presents another opportunity to leverage the power of CUDA. Using different block and grid dimensions, each kernel's execution could be customised to use a variety of thread and block level parallelism; for instance N threads could work in parallel within each block to generate the A and B matrices for solving the PSD of each permutation[15], then individual threads would solve that system, and subsequently calculate the Lagrangian for that permutation, returning a vector of Lagrangian values such that the host process can take the maximum index from that vector and deterministically regenerate the bit-load that created it, as well as retrieving that permutations PSD from the device, completely removing the CPU from PSD generation.

3.4.3 Greedy: Avenues of parallelisation and problem decomposition schemes

Due to the incremental step-wise operation of Greedy bit-loading, and after thorough consultation with Dr McKinley, there is no efficient way to parallelise Greedy without fundamentally changing the operation of the algorithm. To demonstrate this, a 'best-attempt' parallelisation was made, focusing on the updating of tonal costs. After the first step in the algorithm, this is exclusively done on a per-tone basis, meaning that at most, N values are being generated at once. This produced a slow-down of 23% when compared to the CPU-bound algorithm against a 6-line bundle, predominantly due to memory transfers between the host and the device.

As such, Greedy must be disregarded from this project, but will be used as a comparison to answer the question 'Can Massive Parallelism compete with algorithmic improvement?'[16].

3.4.3.1 ISB: Avenues of parallelisation and problem decomposition schemes

ISB, even though its very close in structure to OSB, presents an interesting predicament; while it is still the power optimisation that is the major workload, the incremental operation of this section of the algorithm makes it initially quite difficult to see how to efficiently parallelise it. At first blush, the same patter as OSB

[15] As Derived from equation (2.20)
[16] Without going any further, I would be confident that this is false for greater than 6 lines; Greedy is just that fast

could be followed; where independent channels are separately computed independently, allowing for simple multi-device distribution, or as mentioned, to iterate over the incremental bit combinations of the entire bundle, excluding 'simple' multiple device execution.

In the former case, the style of Figure 35 would be adopted, where each kernel invocation could iteratively perform optimisation on each particular channel. The difficulty is that this algorithm could not really be parallelised any further due to the incremental nature of ISB. It is possible that this could be split up by each user attempting their own bit-load permutation individually, with a record of 'best' bit-load shared between threads in a block, but this is a fundamental break in the ISB algorithm, so would not be guaranteed to be either near-optimal, or even converge at all.

The second option appears to be the most viable, if (at first glance) less applicable to multiple devices. Using an iteration construct like Figure 36, each thread-block could perform each channel's line-loop optimisation. This would only involve a constantly defined loop within the CUDA kernel, which is significantly more performant than a non-deterministic convergence condition as would be required in the former case. In short, this structure would perform channel and permutation parallelism, with each block containing 16[17] threads. While this is not a huge number of threads, it's enough to sufficiently occupy the device. Additionally, CUDA's shared memory space can be used such that at each loop, a block-shared store of the running-bit-load would be updated on each per-line optimisation, containing the bit-permutation with the highest Lagrangian. With 224 ADSL channels, there is no reasonable condition under which this would require more than one device (Number of Threads $= B_{\text{max}} \times K$), but if desired, the channel range could be partitioned across devices.

While the first option will be explored, but the second will be the focus of most development time.

3.4.3.2 Development of generalised GPU workload sharing model

One of the most significant drawbacks of CUDA is its computational simplicity; that is to say that CUDA has relatively little workload partitioning and runtime optimisation when compared to systems such as OpenCL or OpenMP. As such, and from the insights found previously in this section, a generalised workload model was developed to produce near-optimal grid and block dimensions for generic kernels. Although these design patterns are configured with the computation of single PSD/Lagrangian calculations in mind, the same theories of block, warp, grid and device partitioning can easily be applied to any computing problem on GPU's[18].

Each CUDA device has an optimal occupancy ratio for a particular kernel, involving the number of SP's available on the device, expected memory accesses, etc., and as such these patterns will not be perfect. They are simply 'quite good', and attempt to dynamically assess optimal block and grid assignments based on individual hardware configurations without having to inspect the kernels being executed. The first of these patterns is a per-device workload calculation. This queries the device for information such as the number of SP's, number of threads per warp, and the maximum permitted threads per block [19].

Given a maximum value of 'workload', in this case the number of thread executions ideally desired, the number of warps that this workload requires is calculated. This is scaled to the number of threads per warp and rounded up to the nearest multiple of the warp size (usually 32) to give an optimal 'thread per block' count. This value is then used, along with a maximum thread-count ceiling, termed 'gridmax', to find the

[17] The number of possible bit-load combinations for the line currently being looped over within the kernel
[18] as long as those problems are independent of each-other
[19] Note, this is not the optimal number of threads, just a ceiling value

optimal number of blocks to execute these threads within while staying within device limits. The python code for this function is shown in figure 40. Note that this code is developed for 1-D grids and blocks, but could easily be extended for multi-dimensionality or for cooperative thread execution.

```
    def _workload_calc(self,workload):
        warpcount=((workload/warpsize)+(0 if ((workload%self.warpsize)==0)else 1))
        warpperblock=max(1,min(8,warpcount))
        threadCount=warpsize * warpperblock
        blockCount=min(self.gridmax/threadCount,max(1,(warpcount/warpperblock)+(0 if ((warpcount%↩
            warpperblock)==0)else 1)))
        return (threadCount,blockCount)
```

Figure 40: Sanitised Python function for near-optimal Grid and Block dimensions for thread-independent operations

3.4.3.3 Development of generalised multi-device function queue

A GPU device and a host process must be paired, i.e., one host-bound process can execute CUDA kernels on only one GPU device [20]. This restriction, coupled with Python's Global Interpreter Lock (GIL), which is a mutex that prevents multiple native threads from executing Python byte-codes simultaneously[21], requires that for multiple GPU devices to me leveraged, additional multi-processing structures must be used.

There are two major families of multiprocessing structures in Python; one that leverages system threads, like Unix pThreads, in a module called 'threading', and a second that uses full processes, akin to OpenMP. The threading model was selected as the use of multiple processes requires that the entire application (or at least the sections of the application that must deal with multiprocessing) be immutable. This disallows the use of class instance methods (such as all of the bundle and algorithm class functions). Further, processes are significantly 'larger' in terms of memory allocation, and significantly slower in terms of process forking, when compared to the lightweight threading model.

As such, the generated GPU class, was augmented with a secondary class of persistent GPU threads. These objects, once instantiated with a GPU device index on application start-up, wait on work items to be put one a queue by the GPU parent class. Once a work item is received, the appropriate internal method for each algorithm is selected and executed on the GPU to which it has been instantiated. The advantage of this queueing method is that work is inherently balanced across devices without any declared load-balancing algorithm since as soon as each thread has completed its current work item and returned its results to the parent class, it simply picks the next free item from the queue.

The overheads incurred in this process are insignificant, and generally hidden; since CUDA executions are none blocking (i.e. unless told explicitly to wait, a host process can continue to process other information during kernel running time), and that these threads are persistent. This persistence is important, since in the single GPU model, each method execution the GPU device must be reinitialised every time the method is called. With this threading model, each device is initialised once during application start-up, and this initialisation time can be hidden behind other work being done such as the generation of the bundle channel matrix.

[20] This has changed in CUDA 4.0 RC2, but due to PyCUDA's current lack of direct hooks into this functionality, this option could not be explored sufficiently for inclusion in this document

[21] This is required due to Python's memory management system not being thread-safe

Further, this threading model allows the application to be device and system agnostic; automatically adapting to a single, twin, or quad-device system with no user action. This combined with the previously mentioned near-optimal workload calculation could even allow for mixed device systems[22].

3.5 GPU Solutions and Verification

The solutions arrived at for GPU consist of a series of C-sourced CUDA kernels, along with a collection of device-side functions. While PyCUDA exposes some high-level GPU abstraction capabilities, such as parallel per-element array manipulations, this avenue of work could not be applied cleanly to the problem set, mainly due to the need for intra-element operations such as PSD solving.

Before diving into details; the OSB solution consists of three separate per-channel kernels that replicate the operation of 'optimise_p' in the serial implementation (Bit-load/AB generation, PSD Solving, LK calculation), executing for all bit-load permutations simultaneously; while ISB consists of a single kernel that performs per iterative optimisation across a range of channels simultaneously, with the kernel being iteratively executed until bit-load convergence[23].

Additionally, a secondary function was generated for global verification of 'optimal' bit-load PSD's, used during final recalculation of per-line power ratings.

These kernels and device functions are shown in detail in Appendix E

3.5.1 OSB GPU

In the OSB solution, the full permutation range is sliced based on the maximum efficient capacity of the device being used; this generally allows for a maximum three-line solution to be found in 'one shot', but since OSB permutations are computationally explosive, often the kernel triplet is executed many times to span this permutation range to arrive at an optimal solution for a single channel.

This kernel triplet consists of:

- 'lk_osb_prepare_permutations': Generate A and B matrices for PSD calculation of all bit permutations
- 'osb_solve': Solve all permutation's PSD's in parallel
- 'lk_max_permutations': Solve all permutation's PSD's in parallel

The reason for the separation is partly logically and partly technical; the generation of A and B matrices is made parallel in both the permutation and line ranges, as each A/B generation shares many values for each respective permutation. Therefore the block and grid dimensions for CUDA execution were set so as to enhance memory coalescing and column-major shared memory accesses; each 'block' calculates one permutation, and each thread within the block calculates one row of A and B. The Solving and LK generation

[22] Although this functionality has not been tested, since no systems were available for test operated under a mixed-device configuration

[23] Strictly speaking, these calculations are not simultaneous, as individual warps are swapped in and out of the SM's, but for the sake of simplicity, 'simultaneously' is used

kernels do not easily exhibit this level of parallelism and as such are executed under a thread-share model; i.e. each thread is independent but grouped as a 'best attempt' at efficient memory access.

As this algorithm operates on a per-channel basis, the major input data (the cross-talk gain matrix) can stay resident on the device for the complete execution of one channel, reducing the time spent transferring data to and from the device. Additionally, this algorithm allows access to not only the optimised bit-load, but also to the PSD of that bit-load, which remains resident in device memory. This reduces repeat-calculations.

After each kernel execution, as CUDA has no rational way to perform linked-reductions (such as argmax), the device array containing LK values is returned to the host, and the maximal LK-permutation found from the array-index. If this LK value exceeds the current global maximal LK, then the PSD for that value is also pulled from the device, and returned to the calling function.

The multi-device extension to this implementation is a simple queue-based channel distribution across devices, as discussed previously. An attempt was made to distribute permutation slices instead of channels, but due to excessive memory redundancies (i.e the same cross-talk gain matrix having to be constantly sent to the host) this implementation was significantly slower that the channel-distributed model[24].

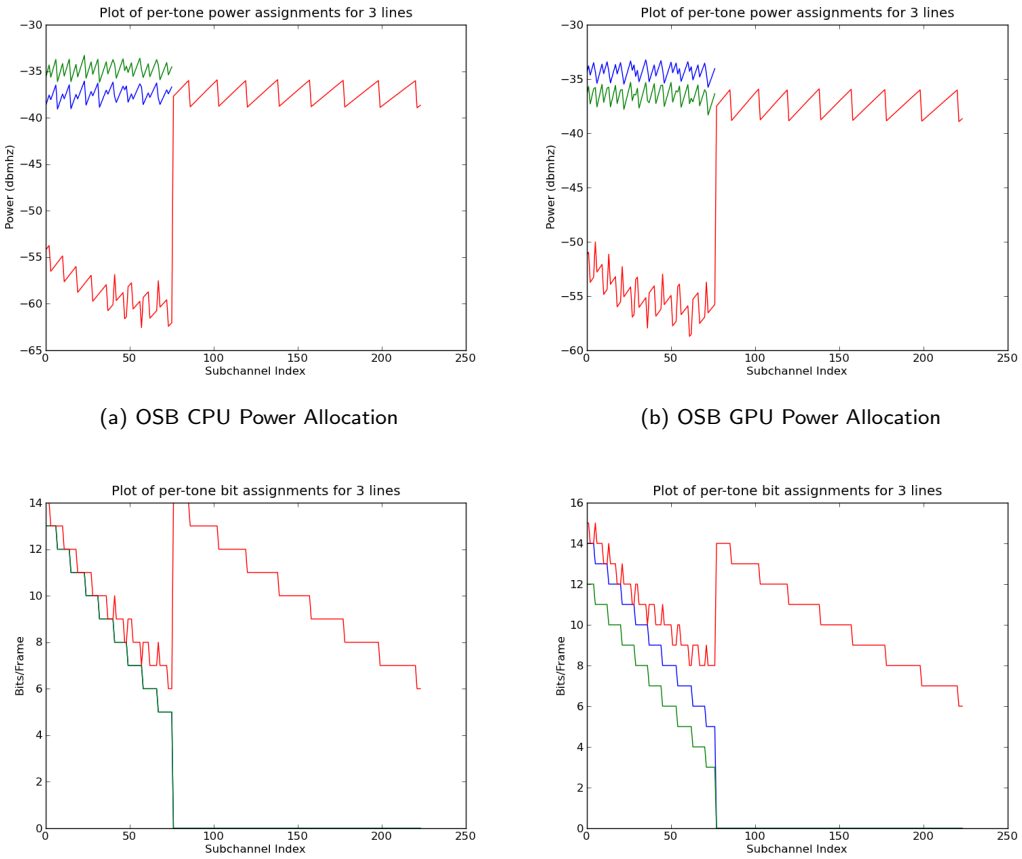
3.5.1.1 Verification

The natural verification method is to compare returned PSD's against the previously-verified CPU implementation (Figure 41). What is interesting about this result is that while power allocations match almost perfectly, bit allocations differ, it appears, significantly. This is due to the inherent numerical instability of the Lagrangian maximisation method. In relative terms, single-differences in bit allocations correspond to minimal PSD changes, and the absolute difference in total bit-load between CPU and GPU versions is less than 35 bits-per-frame over the entire bundle.

As this is an investigation into the viability of GPU implementations and not a study in their correctness, it is felt that this is an acceptable margin of error.

This verification stands for both single and multi-GPU executions.

[24] For four line OSB, the permutation sharing implementation encountered a versus-CPU speed-up of 3x, compared to, as we'll see later, a channel-distributed speed-up of 10 on a single GPU



(a) OSB CPU Power Allocation

(b) OSB GPU Power Allocation

(c) OSB CPU Bit-load

(d) OSB GPU Bit-load

Figure 41: Comparison of three line, near far scenario OSB: CPU/GPU

3.5.2 ISB GPU

In the ISB solution, the host thread iteratively calls a single kernel that operates on all channels simultaneously, 'isb_optimise_inc', that increments across each line and updates a shared 'optimal' bit-load for that iteration. The kernel is repeatedly called until the bundle/sub-bundle bit-load has converged[25].

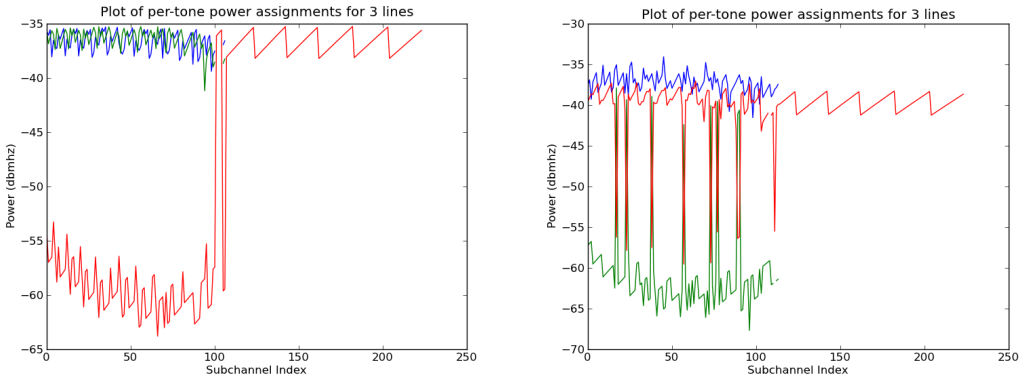
This kernel operates on a mixed-thread-share model, where each block within the CUDA grid calculates the optimal load for a single channel, and each thread within the block performs the PSD and LK calculations for the line being iterated upon. As such, the 1-D grid takes the form $[K|B_{\text{max}}]$. Each block maintains a shared memory location for the 'best' bit-load permutation from each successive line-round, and thus collects an 'optimal' bit-load across all lines on that channel. As such, strictly speaking this kernel is not limited by the number of lines, but by how long it takes for the bit-loads to converge [26]

[25] Sub-bundles in the case of multi-GPU operation

[26] Experimentally it has been found that this implementation of ISB can happily operate on at least 16 line bundles within a practicable time frame(24 seconds), and the only reason for not going further than that is the constraints of 32-bit numerical representation of bit-permutations

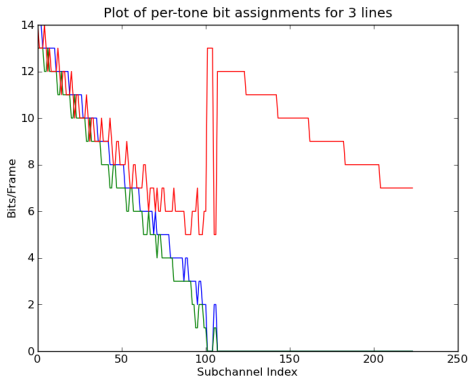
3.5.2.1 Verification

Again, the simplest way to verify operation is direct comparison to CPU bound implementations (Figure 42) In this case the graphs tell a very different story than that of OSB; as can be seen, the GPU implementation has power convergence problems between the green and red lines, but the bit-loads applied are nearly a perfect match. This is again due to numerical representation problems, where by the PSD/LK calculations produce vanishingly small differences between these two line segments, leading to oscillation of bit-load allocations across these lines. Again, the end result of this is a global bundle rate that is within 3% margin of error between CPU/GPU implementations.

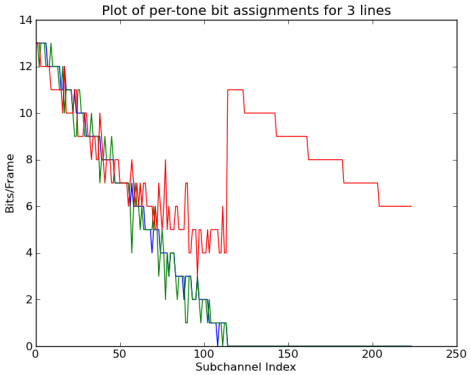


(a) ISB CPU Power Allocation

(b) ISB GPU Power Allocation



(c) ISB CPU Bit-load

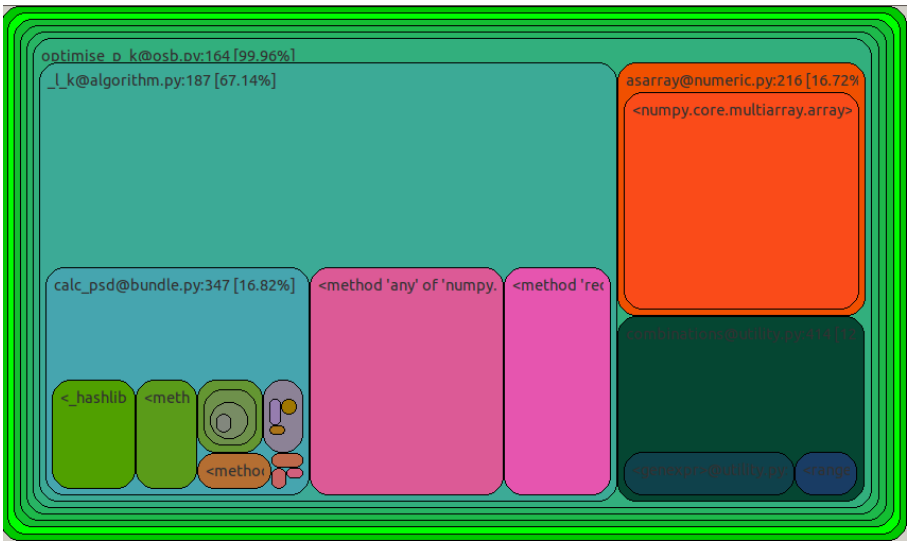


(d) ISB GPU Bit-load

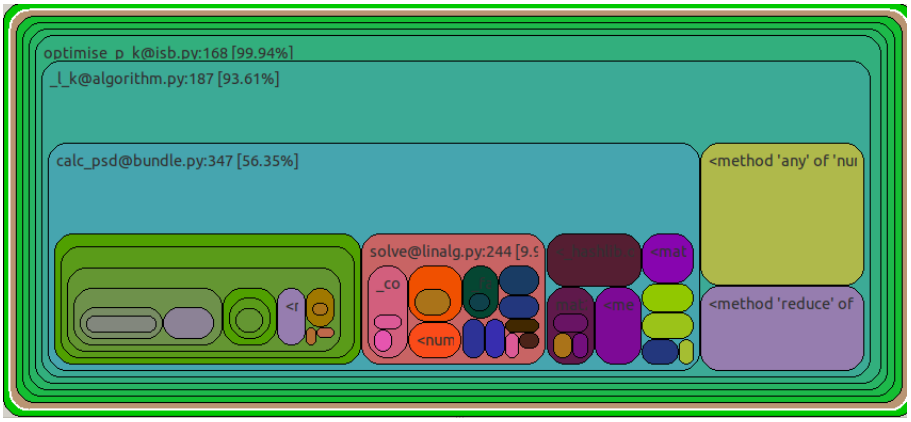
Figure 42: Comparison of three line, near far scenario ISB: CPU/GPU

3.5.3 MIPB GPU

No MIPB solution attempted was functional; as stated, MIPB as it stands has no direct parallelisation vector. Work is under way to perform parallel pre-computation of bit-load chains, but as this is a fundamental divergence from this reports aim, it is not within the scope of this discussion.



(a) OSB Profile SquareMap



(b) ISB Profile SquareMap

(c) MIPB Profile SquareMap

Figure 39: SquareMap output from RunSnake showing percentage execution time for a four line bundle scenario

Chapter 4

Results and Performance Analysis

4.1 Introduction

The format of this chapter is as follows:

- Discussion of Verification results and implications
- Discussion of GPU OSB performance and line / device scalability
- Discussion of GPU ISB performance and line / device scalability

From the results of solution development, it's clear that while computational accuracy has been a casualty of the move to GPU, speed has been greatly improved. This section attempts to quantify this speed-up and the factors that contribute and detract from it.

In order to quantify this speed-up, it is important to look at three main factors; runtime execution in terms of bundle size, runtime execution in terms of multiple device use, and comparative speed-up characteristics compared to CPU execution.

These results will be presented graphically and in tabular format.

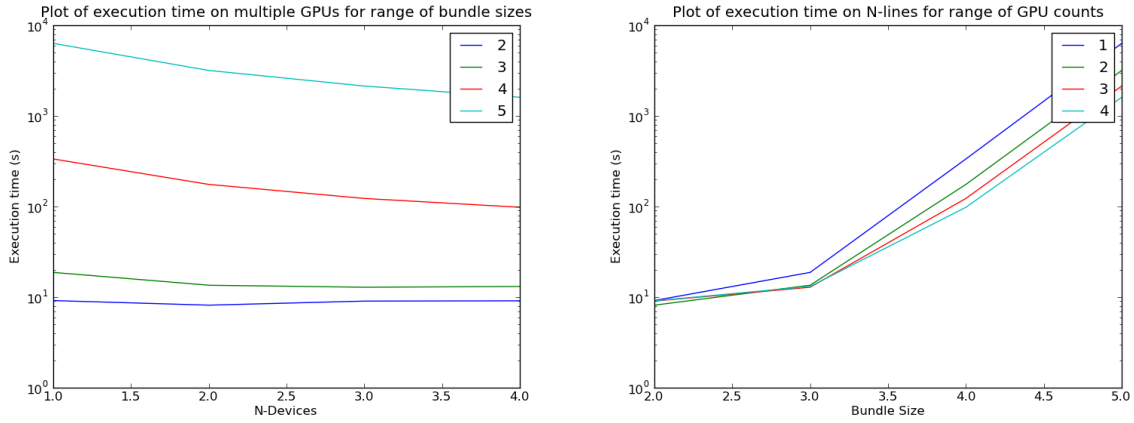
4.2 OSB GPU Performance and Scalability

4.2.1 Runtime vs Bundle Size vs Device Count

These results largely reflect what would be expected of parallel performance behaviour, specifically Amdahl's Law; as the number of parallel execution units is increased, the execution time asymptotically drops to a level beyond which increased parallelism is fruitless. This is particularly clear in figure 44a for the lowest bundle sizes, where it appears that after using only two devices, no further gains are attained. This can easily be explained given the relatively small workload experiences at these sizes of bundle. As the bundle size is

N	CPU time	GPU Count runtime (s)			
		1	2	3	4
2	73.881	9.159	8.147	9.036	9.097
3	651.977	18.737	13.523	12.851	13.100
4	3118.691	334.820	175.373	122.746	98.096
5	16597.065	6320.587	3175.932	2136.499	1605.589

Figure 43: GPU Performance analysis of OSB: execution times against bundle sizes & device counts



(a) Analysis of Execution speed vs N-GPUs

(b) Analysis of Execution speed vs N-Lines

Figure 44: Graphs of OSB Multi-GPU performance

increased, the increase in workload allows multiple devices to be more useful, as the execution time of one devices work packets overtakes the memory transfer and serial algorithmic overheads.

Also indicated in both charts is that the two-device results indicate a 'sweet spot'; where the most speed-up is attained for the least device-count. For larger bundle counts, this is not applicable, as the workload is significant enough to allow for efficient work allocation taking advantage of all devices.

It is also evident from these results that this form of parallelism cannot avoid the computation-explosion inherent in OSB.

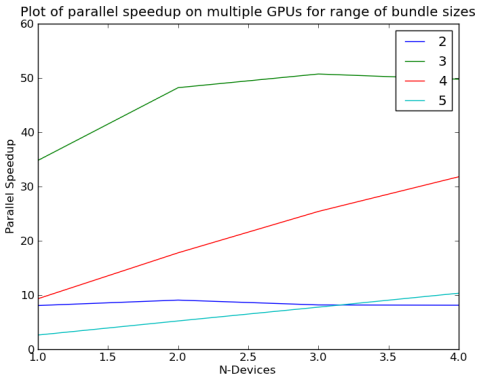
4.2.2 Relative Speed-up

In order to assess the viability of using GPU for this application, it's important to put these results in the context of serial calculations.

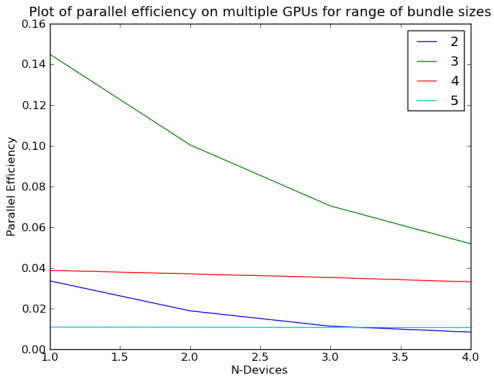
To do so, the Parallel Speed-up ratio is applied to the results, such that $S_p = \frac{T_1}{T_p}$ where T_1 is the execution time for a serial instance, and T_p is the parallel execution time. This is a particularly difficult relationship to apply to GPU; should one count each device as a parallelisation unit, or each SM on each device? This is not so much a problem when calculating the pure speed up of an application, as the number of processing units is not a factor, but is a serious consideration when looking at the parallel efficiency, $E_p = \frac{S_p}{P}$ where P is the

number of parallel processing units.

Initially looking purely at the parallel speed-up attained, the results speak for themselves.



(a) OSB Parallel Speedup



(b) OSB Parallel Efficiency

It is clear again that the expected workload is a major factor; the two line bundle 'bottoms out' very quickly, and its speed-up does not grow with additional resourcing. However, the three line bundle reaches a peak at a speed-up of 50.7x faster than CPU, and starts to drop. This echo's the idea that each workload has a particular 'optimal' processing unit count, and going beyond that count degrades performance.

Looking towards parallel efficiency, strictly speaking each device has 240 PUs[1], so to be fair to the CPU comparison, this is factored into the efficiency calculation thus; $E_p = \frac{S_p}{240 \times P}$ where P is taken as the number of GPUs.

While this looks like an appalling figure, the cost comparison must be made between having 240 PUs sitting on one device in a home computer, versus 240 CPUs stocked in a server farm.

An interesting development appears in the relative performance between CPU and GPU on five-line bundles; even though GPU is still significantly faster than the CPU implementation, the speedup is an order of magnitude less than for lower size bundles. This is due to the increased utilisation of PSD caching in the CPU version, leading to massively increased memory use but 'reduced' execution time. A future development could be the application of dynamic caching techniques to the GPU implementation.

4.3 ISB GPU Performance and Scalability

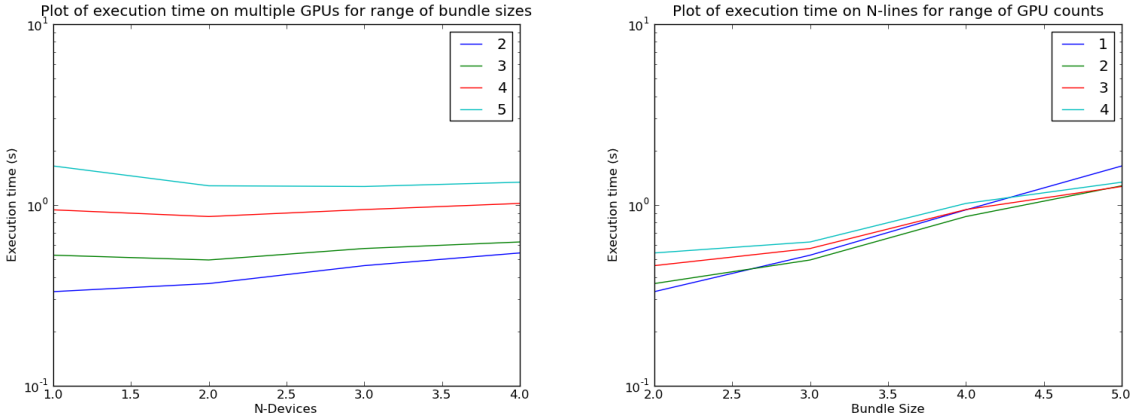
4.3.1 Runtime vs Bundle Size vs Device Count

The results of ISB are particularly interesting; looking specifically at the multi-device behaviour of the two-line bundle, its clear that using more devices for this small workload actually makes execution much much worse. This can again be explained as an artefact of CUDA's memory transfer costs. In-fact in ISB, there is very little justification for applying multiple devices to this problem. This pattern of behaviour is also indicated in the larger-line versions, but shows a 'sweet spot' at two devices.

[1]Each device is a Tesla T10 processor, consisting of 32 SMs with 8 CUDA cores (SPs) per SM

N	CPU time	GPU Count runtime (s)			
		1	2	3	4
2	27.625	0.363	0.000	0.000	0.000
3	76.384	0.528	0.497	0.575	0.625
4	225.278	0.940	0.863	0.944	1.020
5	650.274	1.644	1.277	1.266	1.337

Figure 45: GPU Performance analysis of ISB; execution times against bundle sizes & device counts



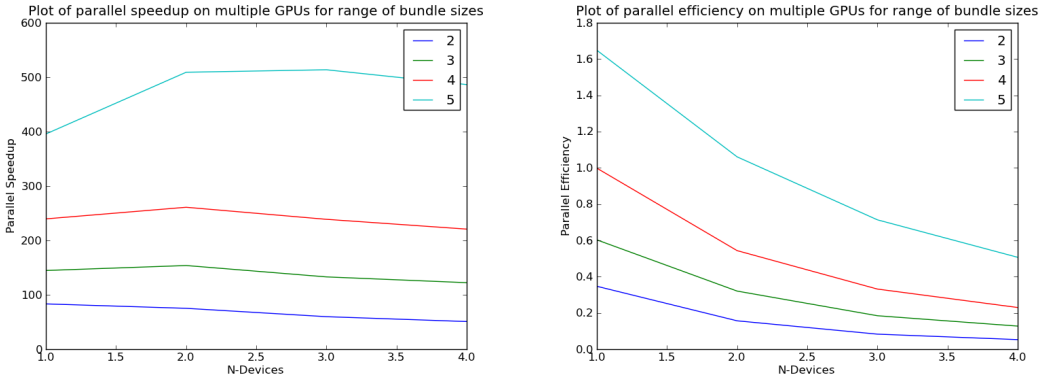
(a) Analysis of Execution speed vs N-GPUs

(b) Analysis of Execution speed vs N-Lines

Figure 46: Graphs of ISB Multi-GPU performance

4.3.2 Relative Speed-up

Applying the same caveats as with OSB, GPU ISB performance relative to CPU execution can be inferred. Counter to OSB, The parallel efficiency curves demonstrated are much more stable. This stability is echoed



(a) ISB Parallel Speedup

(b) ISB Parallel Efficiency

in the more consistent curves of both the speedup graph and N-line execution speed. From this it can be inferred again that there is a partial 'sweet spot' at two devices, but this is not nearly as pronounced.

What is impressive is the level of speedup attained by ISB compared to OSB; since ISB optimises the entire bundle in parallel, there are far fewer memory transfers, meaning that there is less associated cost per kernel invocation. Even comparing like-for-like, the maximum four line speed-up in ISB is almost five times that of OSB's (238x), and ISB's speed-up appears to be tied, at least in this lower-end of the bundle size scale, to the number of lines. Again, this is due to the internal iterative nature of GPU ISB; the more lines there are, the more work each kernel does, the lower proportion of time taken by memory transfers, leading to greatly improved performance over CPU-bound implementations.

Chapter 5

Evaluation and Conclusion

5.1 Comments on DSM

DSL is here to stay, and efficient DSM algorithms must be a part of the evolution of The Internet. Even with the growth of FTTx networks, DSL still plays a major role in last-mile local-loops, especially is sub-urban and rural areas. Considering the complete lack of past-experience in this area, the development of this project was both entertaining, and frustrating; many of the stated DSM algorithms and existing scholarly works do not discuss or even hint at the implementation pitfalls they include, and as such I relied heavily on the consult of Dr McKinley, whose experience in this area is beyond reproach.

In terms of project development, I feel that I got too involved in the generation of the bundle-simulation framework and CPU-bound algorithms, and this put the over all project significantly out of stated schedule, and as such had less time to focus on the problem of GPU optimisation and testing.

5.2 Comments on GPGPU

The results of development and implementation of GPU-bound algorithms show that CPU behaviour can be closely modelled GPU, with impressive performance improvements, but that those improvements come at a price of accuracy, and are far from linear in terms of performance efficiency; as demonstrated in the precipitous falls in efficiency in ISB and OSB when using higher numbers of devices.

The decision to develop a paired PyCUDA-CUDA implementation was a double-edged sword; architectural development was very fast and stayed close to the original structure of the algorithms, but the combination of PyCUDA's own deficiencies in terms of CUDA debugging, poor internal support for Python as a language in general, my own lack of significant experience in either Python or CUDA, and CUDA's inherent abstract complexity, made the entire project harder than it truly needed to be.

What is clear from going through this process is that GPU development, and CUDA especially, is technically challenging. The process of development was hampered at many points; QUB is not currently suitably equipped for advanced development in this area, with several service outages delaying development by weeks,

but even over the course of this project, this state of affairs is improving; new systems are coming online for HPC use; more students are taking advantage of modules such as CSC4005/CSC7007, which includes a CUDA programming component; more and more research clusters are turning to GPU for performance to get over the 'hump' in Moore's Law.

The plummeting price of GPGPUs, and the increasing competition between different manufacturers means that as computation units, GPU's represent a great opportunity for serious investigation into how to leverage their inherent power in a way that is accessible to more than just a select group of CUDA gurus.

In short, I believe that GPGPU has the potential to become a major component of the software engineering domain, but in order for widespread growth, additional programming tools and education are badly required, and as such remains a technical and academic curiosity outside of advanced computation centres. This project has demonstrated that amazing performance gains can be obtained, but at a severe cost of development complexity.

5.3 Comments on Personal Project

My own timekeeping has been terrible, and I have run consistently behind schedule for most of the year. This is due to a combination of biting off more than I could chew (both project, curricular, and extra-curricular), my own technical optimism, and several hardware and software issues throughout the year that seriously slowed down progress, but I believe that the project satisfies the specification laid out for it.

5.4 General Conclusion

The aims of this project were to assess the viability of applying GPGPU programming principals to the field of DSM. It is my belief that GPGPU, and its associated massively parallel programming methodologies, present a technically phenomenal shift in the way such algorithms can be developed and implemented. The performance gains found in even these 'naive' implementations of OSB and ISB show that the use of GPU processing systems in this complex field has a great deal of potential, but there are many many 'kinks' to work out.

5.5 Future Work

With the implementations as they stand, there is much potential for improvements leveraging some of the recently released functionality of CUDA, such as concurrent kernel execution, discrete multi-device operation, and many memory management improvements. To fully leverage these improvements, it would be advisable to re-implement the simulation framework into C++, now that the technical problems are fleshed out, to allow for more finely tuned control of CUDA. Another potential avenue for investigation would be a similar project in OpenCL, which would allow the use of configurable computing devices such as FPGA's and the Tilera architecture, both of which could significantly improve the speed of PSD vector solutions.

The most disappointing aspect of this project was the failure to parallelise MIPB. Dr McKinley agrees that,

after investigation, there is no rational parallelisation of MIPB as it stands, but future work in this area could include a collaboration with Dr McKinley to investigate algorithmic modifications that could be made, generating a brand new algorithm, based on MIPB, that could be suitable parallelised.

References

[1] BT PLC. Results for the third quarter and nine months to 31 December 2010. http://www.btplc.com/News/Articles/Showarticle.cfm?ArticleID=E9EBBBBE-5744-4DF7-B30C-D279879ADA77, 3 February 2011.

[2] Office for National Statistics. Internet Access. http://www.statistics.gov.uk/cci/nugget.asp?id=8, 27 August 2010.

[3] OfCom. UK fixed broadband speeds, November/December 2010. http://stakeholders.ofcom.org.uk/market-data-research/telecoms-research/broadband-speeds/speeds-nov-dec-2010/, 2 March 2011.

[4] C.E. Shannon. Mathematical theory of communication. Bell System Technical Journal, 27(3-4):379–423, 1948.

[5] Peter J. Silverman Thomas Starr, John M. Cioffi. DSL Advances. Prentice Hall, 2003.

[6] J. A. C. Bingham. Multicarrier modulation for data transmission: an idea whose time has come. IEEE Communications Magazine, 28(5):5–14, May 1990.

[7] J. M. Cioffi. A multicarrier primer. ANSI T1E1, 1991.

[8] Various. Orthogonal frequency-division multiplexing: Key Features. http://en.wikipedia.org/wiki/Orthogonal_frequency-division_multiplexing#Key_features, November 10.

[9] Burton R. Saltzberg. Wiley Encyclopedia of Telecommunications, chapter Carrierless Amplitude Phase Modulation. John Wiley & Sons, Inc, 2003.

[10] R.Steele W.T. Webb. Variable Rate QAM for Mobile Radio. In IEEE Transactions on Communications, volume 43, 7 July 1995.

[11] J. Verlinden R. Cendrillon, M. Moonen. Optimal multiuser spectrum management for digital subscriber lines. IEEE International Conference on Communications, 1:1–5, 2004.

[12] Peter J. Silverman Thomas Starr, John M. Cioffi. Understanding Digital Subscriber Line Technology. Prentice Hall, 1999.

[13] Alastair McKinley. Novel Bit-Loading Algorithms For Spectrum Balancing In Digital Subscriber Lines. PhD thesis, Queen's University Belfast, September 2009.

[14] ANSI. Spectrum management for loop transmission systems. ANSI, t1.417-2003 edition, September 2003.

[15] Conexant Systems Alcatel-Lucent, Adtran Inc. Revised 100 pair channel model with asymmetry. In ANSI Contribution NIPP-NAI-2007-183, November 2007.

[16] Sumanth Jagannathan. Interference and outage optimization in multi-user multi-carrier communication systems. PhD thesis, Stanford University, 2008.

[17] K.J. Kerpez. Near-end crosstalk is almost gaussian. IEEE Transactions on Communications, 41(5): 670–672, May 1993.

[18] G. Ungerboeck. Channel coding with multilevel/phase signals. IEEE Transactions on Information Theory, IT-28:55–67, 1982.

[19] I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. Journal of the Society for Industrial and Applied Mathematics, 8:300–304, January 1959.

[20] R. Nilsson F. Sjoberg, M. Isaksson and jesson. Zipper: a duplex method for vdsl based on dmt. IEEE Transactions on Communications,, 54(8):1245–1252, August 1999.

[21] M.N.O. C.M. Akujuobi, J. Shen Sadiku. A new parallel greedy bit-loading algorithm with fairness for multiple users in a dmt system. IEEE Transactions on Communications,, 54(8):1374–1380, August 2006.

[22] Seong Taek Chung. Implementation and Applications of DSL Technology, chapter Dynamic Spectrum Management. Auerback Publications, 2007.

[23] V. Pourahmad J. Cioffi, M. Brady. Vectored DSLs with DSM: The road to Ubiquitous Gigabit DSLs, 2006.

[24] J.M. Cioffi G. Ginis. Vectored-DMT: A FEXT cancelling DSL System. Globecom 2000, 2000.

[25] J.S. Papandriopoulos, J.; Evans. Low-Complexity Distributed Algorithms for Spectrum Balancing in Multi-User DSL Networks. IEEE International Conference on Communications, 2006. ICC '06., pages 3270–3275, June 2006.

[26] Georg Taubock and Werner Henkel. MIMO Systems in the Subscriber-Line Network. In 5th International OFDM-Workshop 2000, 2000.

[27] Wei Yu and J.M. Cioffi. Competitive equilibrium in the Gaussian interference channel. IEEE International Symposium on Information Theory, 2000. Proceedings., 2000.

[28] J.M. Cioffi Wei Yu, G. Ginis. An adaptive multiuser power control algorithm for VDSL. IEEE Global Telecommunications Conference, 2001. GLOBECOM '01., 2001.

[29] Jungwon Lee Seong Taek Chung, Seung Jean Kim. A game-theoretic approach to power allocation in frequency-selective gaussian interference channels. IEEE International Symposium on Information Theory, 2003. Proceedings., June 2003.

[30] H.Levin. A Complete and Optimal Data Allocation Method for Practical DMT systems. IEE JSAC, April 2001.

[31] Wei Yu and R. Lui. Dual methods for nonconvex spectrum optimization of multicarrier systems. Communications, IEEE Transactions on,, June 2006.

[32] M. Moonen P. Tsiaflakis, J. Vangorp. A low complex- ity branch and bound approach to optimal spectrum balancing for digital subscriber lines. In GLOBECOM - IEEE Global Telecommunications Conference,, November 2006.

[33] M. Cendrillon, R. Moonen. Iterative spectrum balancing for digital subscriber lines. IEEE International Conference on Communications, 2005. ICC 2005., 3:1937–1941, May 2005.

[34] J.M. Cioffi. J. Lee, R.V. Sonalkar. A multi-user power control algorithm for digital subscriber lines. IEEE Communications Letters,, 9(3):193–195, April 2005.

[35] G D Cioffi, J M; Forney. Generalized decision-feedback equalization for packet transmission with ISI and Gaussian noise, chapter 4, pages 67–79. Kluwer Academic Publishers, 1997.

[36] A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 1936.

[37] J Von Neumann. First Draft of a Report on the EDVAC. Reporting to United States Army Ordnance Department, June 1945.

[38] David L. McCubbrey Robert M. Lougheed. The cytocomputer: A practical pipelined image processor. In ISCA '80 Proceedings of the 7th annual symposium on Computer Architecture, 1980.

[39] Faye A. Briggs Kai Hwant. Computer Architecture and Parallel Processing. McGraw-Hill Book Company, 1984.

[40] Gordon E. Moore. Cramming more components onto integrated circuit. Electronics Magazine, 1965.

[41] R.K. Zhirnov, V.V.; Cavin. Limits to Binary Logic Switch Scaling–A Gedanken Model. Proceedings of the IEEE, November 2003.

[42] JM Rabaey. Scaling the Power Wall: Revisiting the Low-Power Design Rules. http://www.utdallas.edu/ vojin/classes/EE-7V82/Lectures/Rabey-PowerWall.pdf, November 2007.

[43] Various. Intel Core i7: Sandy Bridge overview. http://en.wikipedia.org/wiki/Sandy_Bridge, April 2011.

[44] M Flynn. Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers, September 1972.

[45] Y. N. Yeh, T.-Y.; Patt. Two-Level Adaptive Training Branch Prediction. In Proceedings of the 24th annual international symposium on Microarchitecture, 1991.

[46] G Amdahl. The validity of the single processor approach to achieving large-scale computing capabilities. In Proceedings of AFIPS Spring Joint Computer Conference, April 1967.

[47] John L. Gustafson. Reevaluating Amdahl's Law. Communications of the ACM, 31(5):532–533, 1988.

[48] Nvidia Corporation. Tesla C2050/2070 Product Brief. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C207 2010.

[49] NVidia Corporation. Compute Unified Device Architecture Programming Guide. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, 2007.

[50] Wen-mei W. Hwu David B. Kirk. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers, 2010.

[51] A Humber. Tokyo Tech Builds First Tesla GPU Based Heterogeneous Cluster To Reach Top 500. http://www.nvidia.com/object/io_1226945999108.html, November 2008.

[52] iVEC. High Performance GPU Computing with NVIDIA, CUDA, and Fermi. http://www.ivec.org/events/2011-01/high-performance-gpu-computing-nvidia-cuda-and-fermi, August 2010.

[53] Kayvon Fatahalian. From Shader Code to a Teraflop: How Shader Cores Work. http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf, 2008.

[54] I Buck M Fatica, P LeGresley. High Performance Computing with CUDA. http://www.cs.bris.ac.uk/ simonm/conferences/isc09/Fatica.pdf, 2008.

[55] NVidia Corporation. CUDA C Programming Guide. http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/ April 2011.

[56] J.E. Dennis Jr; Zhijun Wu. Sourcebook of Parallel Computing, chapter Parallel Continuous Optimisation. Morgan Kaufmann, 2003.

[57] Tomas Nordström Robert Baldemair, Michael Horvat. Proposed Method on Crosstalk Calculations in a Distributed Environment. http://www.nordstrom.nu/tomas/publications/BalEtAl_2003_ETSITM6_033w07.pdf, 2003.

[58] William T. Vetterling William H. Press, Saul A. Teukolsky. Numerical Recipies in Fortran 90. Cambridge University Press, 1992.

[59] Gerard Richter. Rutgers CS510, Numerical Analysis module notes. http://www.cs.rutgers.edu/ richter/cs510/, September 2010.

[60] Charles F. Goulub, Gene H. Van Loan. Matrix Computations, Third Edition. The Johns Hopkins University Press, 1996.

Appendix A: Updated FEXT Cross-talk gain calculation function

```
def calc_fext_xtalk_gain(self,victim,xtalker,freq,dir):
    '''
    Calculate Far End XT Gain
    :adapted from channel_matrix.c
    Uses:
        transfer_func
        insertion_loss
        fext
    (Upstream/Downstream) Cases:
        victim and xtalker == lt
            victim length < xtalker length (4/8)
            victim length > xtalker length (6/2)
            victim and xtalker == length (5)
        victim lt < xtalker lt
            victim nt < xtalker nt (1/9)
            victim nt > xtalker nt (3)
        victim lt > xtalker lt
            victim nt > xtalker nt (9/1)
            victim nt < xtalker nt (7)
        victim and xtalker == nt
            victim length < xtalker length (8/4)
            victim length > xtalker length (2/6)
        victim nt <= xtalker lt (0)
        victim lt >= xtalker nt (0)

    What do the cases mean?
    http://www.nordstrom.nu/tomas/publications/BalEtAl_2003_ETSITM6_033w07.pdf
    "Proposed Method on Crosstalk Calculations in a Distributed Environment"
    In Section 2: Top line is 'victim', A->B is nt->lt


    4 bundle segment types for 2 lines (xtalker and victim);
    H1:xtalker before victim (1,4,7)
    H2:xtalker and victim (all)
    H3:victim after xtalker(1,2,3)
    H4:victim before xtalker(3,6)

    '''
    (vid,xid)=(victim.id,xtalker.id)
    #Check first if there is any shared sector (swapping screws with abs() operation)
    #Check if either v.lt or v.nt is between x.lt/x.nt
    if  (xtalker.lt<victim.lt and victim.lt<xtalker.nt) or (xtalker.lt<victim.nt and victim.nt<xtalker.nt↩
        ):
        log.error("(V/X):(%d/%d):No Shared Sector"%(vid,xid))
        return 0


    if dir == "DOWNSTREAM":                 #If DOWNSTREAM, swap  and negate nt/lt's
        (D1,D2)=(-xtalker.lt,-xtalker.nt)
        (A,B)=(-victim.lt,-victim.nt)
    else: #Upstream
        (D1,D2)=(xtalker.nt,xtalker.lt)
        (A,B)=(victim.nt,victim.lt)


    #Shared length is the H2 Span
    shared_length=abs(max(A,D1)-min(B,D2))/1000.0
    #Head Length is the H1/H4 Span
    head_length=(A-D1)/1000.0
    #Tail Length is the H3 Span
    tail_length=(B-D2)/1000.0

    h1 = self.insertion_loss(head_length, freq)
    h2 = self.insertion_loss(shared_length, freq)
    h3 = self.insertion_loss(tail_length, freq)
    '''
```

```
    This _H takes away the biggest pain of the fext gain cases;
    H1 > 0 in cases 1,3,4,6,7
    H2 active in all cases, but using max(v.nt,x.lt)
    H3 > 0 in cases 1,2,3
    H4 Not relevent for FEXT

    NB, insertion_loss(<0,f)=1

    '''
    H = h1*h2*h3

    try:
        gain = H * self.fext(freq, shared_length)
    except ValueError:
        log.error("Failed on (V/X):(%d/%d):(A,B,D1,D2)=(%d,%d,%d,%d):Shared:%f"%(vid,xid,A,B,D1,D2,↩
            shared_length))
        raise

    return gain
```

Appendix B: OSB Inner PSD Optimisation Function

```
def optimise_p_k(self,lambdas,weights,K,Kmax):
    '''
    Optimise Power per tone, CPU bound version
    '''
    for k in range(K,Kmax):
        log.debug("Launched channel %d search"%k)
        lk_max=-self.defaults['maxval']
        b_max=[]
        #for each bit combination
        b_combinator=combinations(range(self.MAXBITSPERTONE), self.bundle.N)

        for b_combo in b_combinator:
            b_combo=np.asarray(b_combo)
            #The lagrangian value for this bit combination
            lk=self._l_k(b_combo,lambdas,weights,k)
            if lk >= lk_max:
                lk_max=lk
                b_max=b_combo
        #By now we have b_max[k]

        assert len(b_max)>0, "No Successful Lk's found,%s"%b_max
        self.p[k]=self.bundle.calc_psd(b_max,k)
        self.b[k]=b_max
        #print "CPU LKmax %d:%s:%s:%s"%(k,str(lk_max),str(b_max),str(self.p[k]))

    #end for
```

Appendix C: ISB Inner PSD Optimisation Functions (Standard and Loop Exchanged)

```
def optimise_p_k(self,lambdas,weights):
    '''
    Optimise Power per tone, CPU bound version
    '''
    for k in range(self.bundle.K):
        #Convergence value check
        b_this=np.tile(0,self.bundle.N)
        b_last=np.tile(-1,self.bundle.N)
        log.debug("Launched channel %d search"%k)

        #Until convergence of this channels bitload
        while not (b_last==b_this).all():
            b_last=b_this.copy()
            for line in xrange(self.bundle.N):
                lk_max=-self.defaults['maxval']
                b_max=[]
                #for each bit modification
                b_this[line]=0
                while b_this[line] <= self.MAXBITSPERTONE:
                    #The lagrangian value for this bit combination
                    lk=self._l_k(b_this,lambdas,weights,k)
                    if lk >= lk_max:
                        lk_max=lk
                        b_max=b_this[line]
                    b_this[line]+=1

                #By now we have b_max for this user on this channel
                b_this[line]=b_max
            #at this point we are hopefully maximised
        self.b[k]=b_this
        self.p[k]=self.bundle.calc_psd(b_this,k)
    #end while

def optimise_p_k_alt(self,lambdas, weights):
    '''
    Optimise Power: Alternative version used to verify GPU algorithmic loop folding
    '''
    #Convergence value check
    b_this=np.tile(0,(self.bundle.K,self.bundle.N))
    b_last=np.tile(-1,(self.bundle.K,self.bundle.N))
    #Until convergence of these channels bitload
    while (b_last!=b_this).any():
        b_last=b_this.copy()
        for k in range(self.bundle.K):
            log.debug("Launched channel %d search"%k)
            for line in xrange(self.bundle.N):
                lk_max=-self.defaults['maxval']
                b_max=[]
                #for each bit modification
                b_this[k,line]=0
                while b_this[k,line] <= self.MAXBITSPERTONE:
                    #The lagrangian value for this bit combination
                    lk=self._l_k(b_this[k],lambdas,weights,k)
                    if lk >= lk_max:
                        lk_max=lk
                        b_max=b_this[k,line]
                    b_this[k,line]+=1

                #By now we have b_max for this user on this channel
                b_this[k,line]=b_max
            #at this point we are hopefully maximised
    self.b=b_this
    for k in range(self.bundle.K):
        self.p[k]=self.bundle.calc_psd(b_this[k],k)
    #end while
```

Appendix D: Maximal LU Decomposition functions for GPU-based linear system solving

```
#define MAT1 4
#define MAT2 MAT1*MAT1
#define TINY 1.0e-40
#define a(i,j) a[(i)*MAT1+(j)]

#define GO 1
#define NOGO 0

__device__ void d_pivot_decomp(float *a, int *p, int *q){
  int i,j,k;
  int n=MAT1;
  int pi,pj,tmp;
  float max;
  float ftmp;
  for (k=0;k<n;k++){
    pi=-1,pj=-1,max=0.0;
    //find pivot in submatrix a(k:n,k:n)
    for (i=k;i<n;i++) {
      for (j=k;j<n;j++) {
        if (fabs(a(i,j))>max){
          max = fabs(a(i,j));
          pi=i;
          pj=j;
        }
      }
    }
    //Swap Row
    tmp=p[k];
    p[k]=p[pi];
    p[pi]=tmp;
    for (j=0;j<n;j++){
      ftmp=a(k,j);
      a(k,j)=a(pi,j);
      a(pi,j)=ftmp;
    }
    //Swap Col
    tmp=q[k];
    q[k]=q[pj];
    q[pj]=tmp;
    for (i=0;i<n;i++){
      ftmp=a(i,k);
      a(i,k)=a(i,pj);
      a(i,pj)=ftmp;
    }

    //check pivot size and decompose
    if ((fabs(a(k,k))>TINY)){
      for (i=k+1;i<n;i++){
        //Column normalisation
        ftmp=a(i,k)/=a(k,k);
        for (j=k+1;j<n;j++){
          //a(ik)*a(kj) subtracted from lower right submatrix elements
          a(i,j)-=(ftmp*a(k,j));
        }
      }
    }
    //END DECOMPOSE
  }
}

__device__ void d_solve(float *a, float *x, int *p, int *q){
  //forward substitution; see  Golub, Van Loan 96
  int i,ii=0,j;
  float ftmp;
  float xtmp[MAT1];
  //Swap rows (x=Px)
```

```
  for (i=0; i<MAT1; i++){
    xtmp[i]=x[p[i]]; //value that should be here
  }
  //Lx=x
  for (i=0;i<MAT1;i++){
    ftmp=xtmp[i];
    if (ii != 0)
      for (j=ii-1;j<i;j++)
        ftmp-=a(i,j)*xtmp[j];
    else
      if (ftmp!=0.0)
        ii=i+1;
    xtmp[i]=ftmp;
  }
  //backward substitution
  //solves Uy=z
  xtmp[MAT1-1]/=a(MAT1-1,MAT1-1);
  for (i=MAT1-2;i>=0;i--){
    ftmp=xtmp[i];
    for (j=i+1;j<MAT1;j++){
      ftmp-=a(i,j)*xtmp[j];
    }
    xtmp[i]=(ftmp)/a(i,i);
  }
  //solves x=Qy
  for (i=0;i<MAT1;i++){
    x[i]=xtmp[q[i]];
  }
}

__global__ void solve(float *A, float *B, int max){
  //Each thread solves the A[id]x[id]=b[id] problem
  int id= blockDim.x*blockIdx.x + threadIdx.x;
  int p_pivot[MAT1],q_pivot[MAT1];
  if ((GO==1) && (id < max)){
    for (int i=0;i<MAT1;i++) {
      p_pivot[i]=q_pivot[i]=i;
    }

    d_pivot_decomp(&A[id*MAT2],&p_pivot[0],&q_pivot[0]);
    d_solve(&A[id*MAT2],&B[id*MAT1],&p_pivot[0],&q_pivot[0]);
  }
}
```

Appendix E: GPU Kernels and device functions

```
#include <pycuda-helpers.hpp>
#define MAT1 {{matrixN}}
#define MAT2 MAT1*MAT1
#define FAILVALUE {{failvalue}}
#define FPT {{floatingpointtype}}
#define CHANNELGAP {{channelgap}}
#define NOISE {{noise}}
#define MBPT {{maxbitspertone}}
#define TINY 1.0e-40
#define MAXBITPERM {{maxbitperm}}
#define K {{K}}
#define a(i,j) a[(i)*MAT1+(j)]

//Each thread solves the A[id]x[id]=b[id] problem
__global__ void solve(FPT *A, FPT *B){
  int id= blockDim.x*blockIdx.x + threadIdx.x;
  int p_pivot[MAT1],q_pivot[MAT1];
  if ((GO==1)){
    for (int i=0;i<MAT1;i++) {
        p_pivot[i]=q_pivot[i]=i;
    }

    d_pivot_decomp(&A[id*MAT2],&p_pivot[0],&q_pivot[0]);
    d_solve(&A[id*MAT2],&B[id*MAT1],&p_pivot[0],&q_pivot[0]);
  }
}

//Generate A/B for a particular bitload[index]
//Leverages mixed-parallelism using otherline variable
//  i.e can be executed as part of a serial loop
__device__ void generate_AB(FPT *A, FPT *P, FPT *d_XTG, int *bitload, int index, int otherline){
    for (int i=0; i<MAT1; i++){
          //Generate a row of A for this permutation and victim y
          A[index*MAT2+otherline*MAT1+i]=-(CHANNELGAP*((1<<bitload[otherline])-1)*d_XTG[i*MAT1+otherline↩
              ])/d_XTG[otherline*MAT1+otherline];
      }
      //Generate an item of P
      P[index*MAT1+otherline]=(NOISE*CHANNELGAP*((1<<bitload[otherline])-1))/d_XTG[otherline*MAT1+↩
          otherline];

      //Repair an item of A
      A[index*MAT2+otherline*MAT1+otherline]=1;
}

__device__ void lkcalc(int *bitload, FPT *lambdas, FPT *w, FPT *P, FPT *LK, int index){
    FPT lk = 0;
    int broken = 0;
    for (int i=0;i<MAT1;i++){
       //Need to check for negative P's
        if (P[index*MAT1+i]<0)
            broken++;
        lk+=(bitload[i]*w[i])-(lambdas[i]*P[index*MAT1+i]);
    }
    //If anything is broken return a failing value (around -inf)
    if (broken==0)
        LK[index]=lk;
    else
        LK[index]=FAILVALUE;

}

//Given space for A and P, and current_b[N*MAT1] populate P with the psds
//Assume d_XTG is relevant to index
__device__ void d_calc_psd(FPT *A, FPT *P, FPT *d_XTG, int *bitload, int index){
    int i;
    int p_pivot[MAT1], q_pivot[MAT1];
```

```
    //generate A and B
    for (i=0;i<MAT1;i++){
        generate_AB(A,P,d_XTG,bitload,index,i);
        p_pivot[i]=q_pivot[i]=i;
    }
    __syncthreads();
    d_pivot_decomp(&A[index*MAT2],&p_pivot[0],&q_pivot[0]);
    d_solve(&A[index*MAT2],&P[index*MAT1],&p_pivot[0],&q_pivot[0]);
}

__global__ void calc_psd(FPT *A, FPT *P, FPT *d_XTG, int *current_b, int N){
    //Assume we're doing a full channel range recalculation
    int id=blockIdx.x*blockDim.x+threadIdx.x;
    if (id<N){
        d_calc_psd(A,P,&d_XTG[id*MAT2],&current_b[id*MAT1],id);
    }
}
//============================================================================
// OSB FUNCTIONS for set channel execution
//============================================================================
//Generate the A and B for all possible bitloads (in this offset)
//requires grid(MBPT^N,1,1) block(N,1,1)
//thread.y's collaboratively populate A and B for their id
__global__ void lk_osbprepare_permutations(FPT *A, FPT *B, FPT *d_XTG, int offset){
    //Don't need k as its sorted at the host stage for the creation of xtg
    int j=threadIdx.x;
    int myid=blockIdx.x;
    int bitbangval=myid+offset;

    int bitload[MAT1], i;

    //rebase myid to base (MBPT)
    //Unfortunately theres no way around every thread working out its own bitload
    for (i=0; i<MAT1; i++){
        bitload[i]=bitbangval%MBPT;
        bitbangval/=MBPT;
    }
    if (myid+offset<MAXBITPERM){
      generate_AB(A,B,d_XTG,bitload,myid,j);
    }
}

//Solve all A and B psds together.
//requires grid(MBPT^N/threadmax,1,1) block(threadmax,1,1)
__global__ void solve_permutations(FPT *A, FPT *B, int offset){
    int id=blockIdx.x*blockDim.x+threadIdx.x;
    int bitbangval=id+offset;
    int p_pivot[MAT1],q_pivot[MAT1];
    int i;

    //simulate bitload generation for in-place id check, and pivots at the same time
    for (i=0; i<MAT1; i++){
        bitbangval/=MBPT;
        p_pivot[i]=q_pivot[i]=i;
    }
    //Stopper for invalid id's (where bitcombinations is less than maximum blocksize}
    if (id+offset<MAXBITPERM){
        //do the magic
        d_pivot_decomp(&A[id*MAT2],&p_pivot[0],&q_pivot[0]);
        d_solve(&A[id*MAT2],&B[id*MAT1],&p_pivot[0],&q_pivot[0]);
    }
}

//Finally Calculate the LK_Max_permutations
__global__ void lk_max_permutations(FPT *P, FPT *LK, FPT *lambdas, FPT *w, int offset){
    int id=blockIdx.x*blockDim.x+threadIdx.x;
```

```
    int bitbangval=id;
    int bitload[MAT1], i;

    //At this point, B is populated with the P results.
    for (i=0;i<MAT1;i++){
        bitload[i]=bitbangval%MBPT;
        bitbangval/=MBPT;
    }
    if (id+offset<MAXBITPERM){//check for out of range id's
        lkcalc(bitload,lambdas,w,P,LK,id);
    }
    else
        LK[id]=FAILVALUE;
}

//============================================================================
// ISB FUNCTIONS for channel range
//============================================================================

//Do all channels at once for each permutation range
__global__ void isb_optimise_inc(FPT *A, FPT *P, FPT *d_XTG, FPT *LK, FPT *lambdas, FPT *w, int *↩
     current_b, int offset){
    const int k=blockIdx.x;                 //The channel that we're playing with
    const int permutation=threadIdx.x;            //the permutation we're generating
    const int index=k*MBPT+permutation;     //The channel-permutation array index we're building
    int line,i;                         //The internal line loop and generic incrementer

    __shared__ int bitload[MAT1];
    int threadbit[MAT1];

    //copy initial bitload into block memory
    if (permutation < MAT1){
        //current_b.shape(K,N)
        bitload[permutation]=0;
    }
    __syncthreads();

    // This algorithm swaps the k-range and last=this loops
    for (line=0;line<MAT1;line++){
        //copy shared bitload into thread memory
        for (i=0; i<MAT1; i++){
            threadbit[i]=bitload[i]; //Copy into thread memory
        }
        //For this new user, make him special (line-1 should be optimised)
        threadbit[line]=permutation;

        //Solve!
        d_calc_psd(A,P,&d_XTG[k*MAT2],threadbit,index);

        //Calculate lk for each permutation on each channel in parallel
        lkcalc(threadbit,lambdas,w,P,LK,index);

        //Return maxlk bitload for this user on this channel (threadbit partially overwritten, no problem
        bitload[line]=isb_perblock_lkmax_B(LK,NULL,k,(int)MBPT);

        __syncthreads();
    }

    __syncthreads();
    //For each channel, copy bitload back to current_b
    if (permutation<MAT1){
        current_b[k*MAT1+permutation]=bitload[permutation];
    }
    __syncthreads();
    //At the end of this, current_b will contain the optimal bitloads for all channels, addressible as [k↩
         *MAT1+line]
    //    P, addressable as [k*MBPT+bitload] (for the last user)
```

```
    //    lk is more or less useless in this case.
}

//Populate the B array with the max bitload permutation for each block of lk's
__device__ int isb_perblock_lkmax_B(FPT *LK, int *B, int id, int blocksize){
    int i, pmax=-1;
    FPT lkmax=FAILVALUE;
    for (i=0;i<blocksize;i++){
        if (lkmax <= LK[id*blocksize+i]){
            pmax=i;                     //This is the maxlk bitload for this line
            lkmax=LK[id*blocksize+i];
        }
    }
    if (B!=NULL){
        B[id]=pmax;
    }
    return pmax;
}
```