FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Towards GPU Speech Coding

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik

eingereicht von

Axel Goldmann
Matrikelnummer 9607701

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung:

Wien, 29.04.2011

(Unterschrift Verfasser)

(Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Towards GPU Speech Coding

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Media Informatics

by

Axel Goldmann
Registration Number 9607701

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance:

Vienna, 29.04.2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Axel Goldmann
Testarellogasse 24, 1130 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

Thanks to the responsible persons at PDTS Ges.m.b.H Vienna for their supervision during the work, and Univ.Prof. Dr.-Ing. Norbert Görtz for providing some diagrams.

Danksagung

Gedankt sei für ihre Unterstützung bei der Ausführung dieser Arbeit den Verantwortlichen bei der PDTS Ges.m.b.H in Wien, sowie für die Bereitstellung von Diagrammen Univ.Prof. Dr.-Ing. Norbert Görtz.

Abstract

Speech transmission is the central service in many telecommunication infrastructures. The encoding of many channels according to modern standards requires a fair amount of processing capacity. With the recent GPU product lines, powerful platforms have become available as supplement to desktop CPUs. This thesis tries to leverage these developments and examines the possibilities of general purpose GPU employment in the context of speech coding. The speech codec used in the TETRA mobile radio system is implemented using the CUDA programming model. The main question is, how many channels can be encoded in real time on current GPUs. Results show that through careful implementation and with some effort, a substial number of channels can be processed. It seems however that modern multicore CPUs are much better qualified for the task. The presented optimizations are far from complete and further research directions are suggested.

Kurzfassung

Sprachübertragung zählt zu den zentralen Diensten in vielen Telekommunikationsinfrastrukturen. Die Codierung vieler Kanäle gemäs̈ modernen Standards beansprucht einen nicht unerheblichen Rechenaufwand. Jüngste Grafikkartenmodelle bieten eine leistungstarke Unterstützung zu herkömmlichen Prozessoren. Diese Arbeit versucht, jene Entwicklungen auszunützen und untersucht die Möglichkeiten der Grafikkartenanwendung im Kontext von Sprachkodierung. Dafür wird der Codec aus dem TETRA Standard mittels der CUDA Programmierschnittstelle implementiert. Die entscheidende Frage dabei ist, wieviele Kanäle auf gängigen Karten in Echtzeit berechnet werden können. Die Ergebnisse zeigen, das̈ durch sorgfältige Schritte und einigen Aufwand eine erhebliche Zahl an Kanälen von Grafikkarten übernommen werden können. Moderne Mehrkernprozessoren scheinen dennoch besser für die Anforderungen geeignet zu sein. Die hier beschriebenen Optimierungen sind bei weitem nicht erschöpfend und weiteren Untersuchungen ist Potential beizumessen.

Contents

1 Introduction 1

1.1 Motivation 1
1.2 Problem Statement 2
1.3 Contributions 3
1.4 Thesis Structure 3

2 Speech Coding 5

2.1 Speech Coding Motivation 5
2.2 Speech Signal Properties 7
2.3 Speech Production Model 11
2.4 Linear Predictive Coding 11
2.5 Short-term and Long-term LPC 13
2.6 Vector Quantization 19
2.7 Analysis by Synthesis 22
2.8 Code Excited Linear Prediction 23
2.9 Specific characteristics of the TETRA codec 28

3 GPU Properties 33

3.1 Architectural Overview 34
3.2 Streaming Multiprocessors 35
3.3 Device Memory System 37
3.4 The CUDA Programming Model 38
3.5 Discussion 42

4 Related Work 45

5 Implementation 47

5.1 Encoding Procedure 47
5.2 Leveraging GPU Properties 50
5.3 Exploiting Parallelism 51
5.4 Unit Ranking 53
5.5 Selected Units 54

6 Overall Results 65

7 Conclusion **69**

Bibliography **71**

CHAPTER 1

Introduction

1.1 Motivation

As demands for computing power continue to increase in many fields, the shift towards parallel architectures on the commodity market is proceeding. GPUs evolved from specialized circuitry to fully programmable accelerators of non-graphics applications. With the recent transition from pipeline based to universal programmable designs, their adoption has become mainstream. Figure 1.1 shows the trends in floating point performance of GPUs and CPUs to make clear why general purpose GPU usage (GPGPU) became increasingly popular in recent times. For some, acceleration by GPU is even seen as a revival of vector computing, [35].

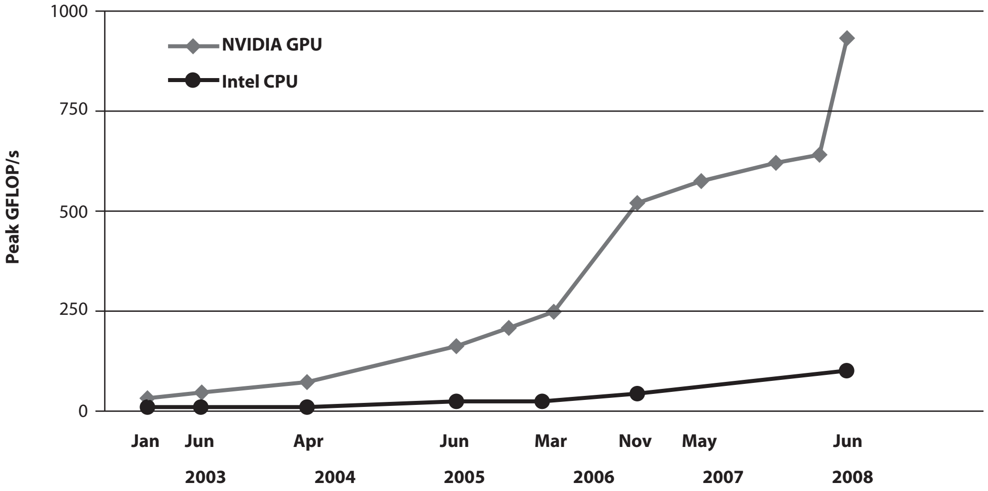

Figure 1.1: Recent trends of GPU and CPU FLOPS, traced and modified from [40].

Until recently, the performance challenges for desktop CPUs were met mainly by increasing clock frequencies [27]. A number of design related restrictions, however, have brought an end to this condition. These so called walls stopped the further pursuit of the previously successful practice of raising single-thread performance [15, 52].

- Clock frequencies cannot be increased anymore without unjustifiable raise of chip heat, power dissipation and leak voltage.
- Exploitation of instruction level parallelism has reached its limits.
- Bus width between CPU and memory cannot further be extended.

At around 2005, commodity CPUs stopped to gain their periodical performance bonus by clock frequency increases. Until that point, innovations in alternative specialized architectures would quickly be negated by this effect. Since then, CPUs became multi-core. Furthermore, CPU and GPU architectures show signs of convergence, which justifies the revision of established algorithms under the aspect of parallel processing.

The motivation behind this thesis is trying to exploit these commodity market trends for telecommunication applications. Modern speech encoding algorithms (such as G.729 used in IP-telephony), require a fair amount of processing capacity. In many networks providing speech services the need to encode or decode many speech channels in a single hub exists. Traditionally, DSP processors on expensive specialized hardware are deployed for this purpose. In this work it should be investigated whether it is possible to substitute specialized infrastructure with commodity hardware, and if there is any benefit from it. The economic idea behind this is to gain from the strong market forces, driven mainly by the computer game industry. Thus, powerful hardware is available at low cost.

GPGPU has already been applied for countless applications, however the field of audio signal processing is left relatively unaffected. And even if current technology might be immature for DSP purposes, this work should be seen as an evaluation allowing to better estimate the potential of further developments in the future.

Speech coding includes many subproblems which are interesting to study under the aspect of parallel processing.

1.2 Problem Statement

The context of this thesis is a private mobile radio infrastructure. It is compliant with the controversial ETSI specification EN 300 392, defining the widely employed Terrestrial Trunked Radio System (TETRA). Targeted at public service institutions, such as police, fire departments or ambulance, the system is designed to provide reliable speech transmission for its subscribers. As a modern standard, its speech transmission method is digital for reasons of bandwidth economy and uses the according TETRA voice codec specified in [22]. The search for an economic unit capable of processing large numbers of speech channels led to the idea to run the TETRA codec on a current streaming platform (GPU). The central question therefore is:

how many channels of the TETRA speech codec can be encoded in realtime on current standard graphic cards?

As a restriction, only cards of the Nvidia family should be considered, and the implementation be carried out using Nvidia's CUDA programming model.

1.3 Contributions

To the best of the author's knowledge this is the first attempt of running a speech codec on a streaming platform. An assessment is given of how many channels a modern GPU may process in realtime. Many GPGPU applications count fully on problem-inherent data parallelism. In this work instead, ways are discussed of how parallelism can be drawn from the simultaneous processing of independent speech channels. The main effort lies in the analysis of the encoding algorithm regarding its mapping to the parallel platform for a single channel. Based on the results of the single-channel implementation is a projection of channel capacity for GPUs in multi-channel operation. Limited input data and complex processing makes speech coding an extremely compute-bound[1] application.

Certain operations necessary for encoding, in particular the handling of short-tap filters and convolution, are discussed for GPU implementation. Thereby, the potential of algebraically transforming problems for better adaptability to the parallel platform is pointed out.

1.4 Thesis Structure

The document is structured as follows. In Chapter 2, a theoretical background for a better understanding of the subject of speech coding is given. After a short introduction to speech coding a detailed illustration of the employed encoding procedure follows. Without this knowledge, the given task in this thesis would not have been feasible. How the streaming platform can be exploited in the implementation requires the knowledge of its architecture, properties, and restrictions as well as the according programming model, which are all given in Chapter 3. Chapter 4 is dedicated to related work and tries to identify existing research activities that, under different aspects, share similar aims with this thesis. Next, Chapter 5 is the description of own contributions and results from implementation efforts and problem analysis. The conclusion in Chapter 7 finally has the purpose to assess the results and discuss future prospects.

---

[1] See [30] for an explanation of the term in the GPGPU context.

CHAPTER 2

Speech Coding

In the following, the theoretical background for this thesis is given. The chapter starts with an introduction to speech coding and properties of digital speech signals. Then all concepts required for the understanding of a modern speech coder are introduced step by step until the final structure is revealed. A description of the specific features of the coder treated in this work closes the chapter.

2.1 Speech Coding Motivation

Speech transmission remains an important component in telecommunication services. About 40 years ago, transformation from the analog into the digital representation of speech signals began. Digitalization introduces a number of convenient effects for signal handling. Besides infinite waveform regeneration, the application of error correction codes, security measures such as encryption, as well as the integration into related systems were all enabled by that transition. Traditionally, analog telephone networks transmitted speech signals restricted to the 300 to 3400 Hz band. For digital systems this speech bandwidth was retained. Signal discretization is achieved with a sampling rate of 8 kHz which ideally allows for maximum frequency components of 4 kHz according to the Sampling Theorem, and in practice is limited to 3.4kHz due to filter margins. Speech signals in this frequency range are called *narrow-band* speech signals and yield sufficient audibility for telephone conversations. Other standards with higher sampling resolution exist but will not be treated in the following. Therefore, 8 kHz will be the assumed sampling rate during all considerations [24], [18].

In its raw unprocessed form, a digital speech signal is in so-called PCM (Pulse Code Modulation) format. That means subsequent signed values represent the analog amplitude as measured during A/D conversion. Thereby, a resolution of 16 bit per sample is required to adequately represent the original waveform for an acceptable listening experience. With 16 bit samples taken at a rate of 8 kHz, the resulting accumulating data totals 128 kilobyte per second and thus PCM makes digital transmission far inferior in terms of bandwidth efficiency compared to its analog predecessor. Therefore, methods of data reduction (or compression) are so important.

First and foremost, this is achieved through *quantization*, which reduces the number of bits representing a sample. A quantization is defined by the partitioning of the whole amplitude range into a number of intervals. After discretization, the obtained amplitude values are then classified according to the chosen range partition. Each sample is represented by its interval. The simplest variant is with equally sized intervals. In practice, the layout is optimized according to application. For speech signals, the interval layout is adjusted to the amplitude density distribution in speech, seen in Figure 2.1, [24], [18].

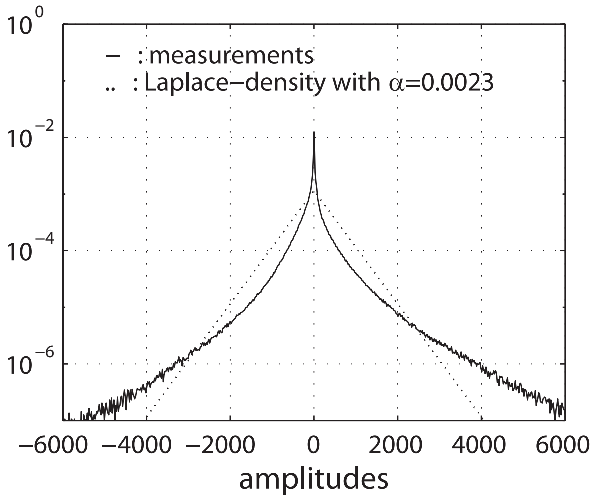

Figure 2.1: Amplitude density distribution of average speech, courtesy of [18].

The average occurrence of amplitudes in speech approximately matches the Laplace distribution which falls off in logarithmic manner around its mean (zero in the case of speech). As an example, the approach of Logarithmic PCM leverages this property: The amplitude intervals are laid out in a way so that samples with rather high magnitudes are assigned relatively less resolution than those of smaller amplitudes. Wider intervals cover more sample values, and hence yield larger errors. On the other hand, values near zero are represented by smaller intervals and after quantization, remain distinguishable to a greater extent. That means, errors are diverted to less common amplitudes. With 8-bit words for the encoding of the quantization levels, the effects of the introduced errors are negligible to the listener. As a measure for quantization performance serves the signal to noise ratio (SNR)

$$SNR = \frac{\sigma_S^2}{\sigma_N^2}$$

That is the relation of signal and error (or noise) powers[1].

Although the amount of data to transmit is reduced greatly with an 8-bit quantization, the resulting rate of 64 kb/s is still too high for many applications involving bandwidth-limited channels such as satellite and mobile communication systems.

---

[1] Speech signals have zero-mean value. Therefore, variance σ^2 is identical to power [18].

Today, even standards with a bit rate of only 16 kb/s are referred to as high bit-rate encoding schemes. Table 2.1 [11], gives an overview of bit-rate classifications for speech codecs.

Category	bit-rate [kbps]
high	>15
medium	5 to 15
low	2 to 5
very low	<2

Table 2.1: Classification of bit-rates for speech coding standards [11].

For comparison, the TETRA codec is classed as low bit-rate and produces about 4.5 kb/s [22]. In the past, great efforts were taken devising methods that allow for such a great reduction of transmission data volumes. Early approaches such as the logarithmic PCM method described above try to reproduce the original time domain waveform as closely as needed in a per-sample manner without considering relations among neighboring samples, let alone properties of whole sequences. These so-called waveform-coding methods, due to their universal treatment of signal samples are not restricted to a certain type of application, but perform poorly in the case of narrow-band speech [11, 24].

More complex encoding methods try to express the essence of the signal by means of an adequate model and are called *parametric* coders. Instead of transmitting the quantized samples, the encoder analyses a whole section of the signal at once, taking into account an underlying model of description. The analyzed signal sections are called *frames*. Certain model parameters are thereby extracted to efficiently describe the essential characteristics of the current speech frame. The obtained parameters are then sent to the decoder, which in turn uses them for synthesis of the speech according to the same model. The parameters cause much less data traffic and it is such parametric techniques that enable codecs with low bit-rates as given in the table above. Although resulting in a loss of quality compared to waveform-coding, the gain of reduced data rates outbalances this disadvantage. Certainly, advantages come with a number of difficulties. For example, the parameters are much more sensitive to transmission errors due to their high significance in the model. Furthermore, the complexity of the underlying models prevents measurement of output quality by terms of SNR and evaluation must be carried out empirically in auditory trials, making development and tests much more expensive [32].

The TETRA codec, which is the actual target of the efforts in this thesis, is an example of a highly advanced form of parametric analysis technique. Before revealing the functionality in detail, the explanation of the basic model of speech production and related signal properties are required.

2.2 Speech Signal Properties

In the human body, speech is produced by causing resonance of the vocal tract, a series of cavities located in throat and head. Basically, air is pressed from the lungs against the vocal cords, which in turn release pulses into subsequent resonance chambers. To a certain extent, these cavities' form and expansion are defined by the attached musculature and thereby allow the speaker to shape the tone. In the oral cavity, the arrangement of tongue and teeth articulates the final sound [11, 24, 32]. One evident property of speech signals is that two distinct modes occur during their production, namely that of *voiced* and *unvoiced* sounds. With vocal cords vibrating, air flow from the lungs is periodically interrupted and the vocal tract therefore excited with a series of pulses resulting in voiced sounds at a certain key tone. Unvoiced sounds in contrast are generated with vocal cords letting pass the air unhindered and are of noisy character [11], [18].

Speech recordings taken from dedicated databases were analyzed to reveal their signal characteristics [18]. The average power-density spectrum (PDS) of narrow-band speech is shown in Figure 2.2.

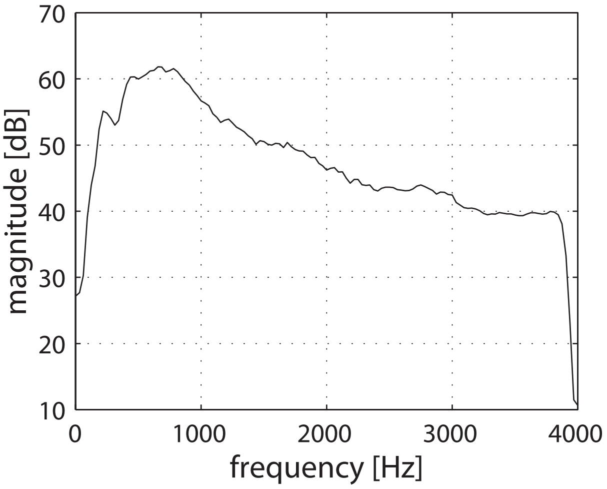

Figure 2.2: Power density spectrum of average narrow-band speech, courtesy of [18].

Its envelope shows low-pass characteristics. This deviation from whiteness implies the presence of correlations between samples. For a digital speech signal $s[n]$, the time-domain equivalent of the PDS, the autocorrelation function (ACF), is given by

$$\varphi_{ss}(\lambda) = \mathrm{E}\left\{ s\left[n+\lambda\right] s\left[n\right] \right\}$$

If constricted to a certain signal section (or frame) of N samples, it is written as[2]

$$R_{ss}(i) = \frac{1}{N-i} \sum_{n=0}^{N-1-i} s\left[n+i\right] s[n]$$

The ACF expresses the occurrence of repeating patterns dependent on a time-lag i or λ respectively. In the average speech signal, the ACF converges to the mean (zero) for lag values

---

[2] For the details of the relation of PDS and ACF see [11].

higher than 20. That means, relations between samples range up to 2.5 ms * 20 = 50 ms in average speech.

Figures 2.3 and 2.4 show the time domain signal, the magnitude spectrum, and the ACF for unvoiced and voiced speech frames with a length of 30 ms, corresponding to 240 samples.

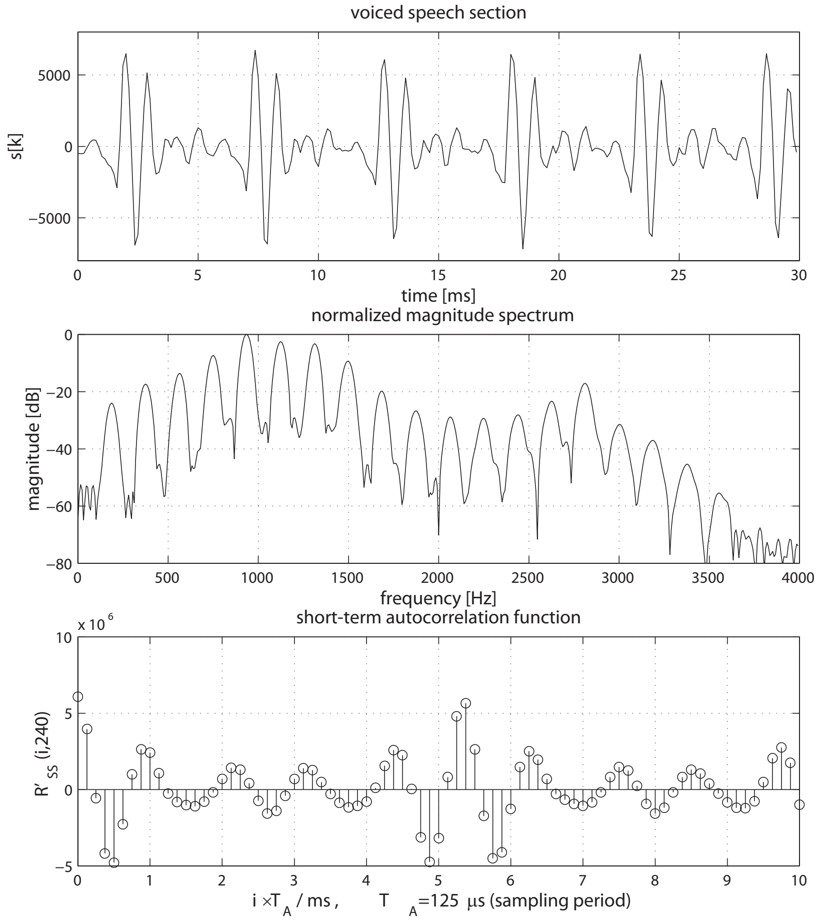

Figure 2.3: Time signal, magnitude frequency spectrum and ACF of a voiced speech frame, courtesy of [18].

In the time-domain plot of the voiced frame the periodicity is clearly visible. The 5.4 ms period corresponds to the 185 Hz key tone indicated by the first local maximum in the associated frequency spectrum plot. The periodicity is responsible for the pitch of voice. It ranges from 50 to 400 Hz dependent on age and gender of the speaker. In contrast, the unvoiced frame shows much less dynamic and also lacks periodicity.

Both displayed frequency spectra are defined by a given configuration of the vocal tract. Thereby, the resonance frequencies of the vocal tract appear as regions of local maxima. The key frequencies at which these maxima occur are called *formants* and are characteristic for each phoneme. In Figure 2.3 for example, two formants show up at 1000 Hz and 2700 Hz. The

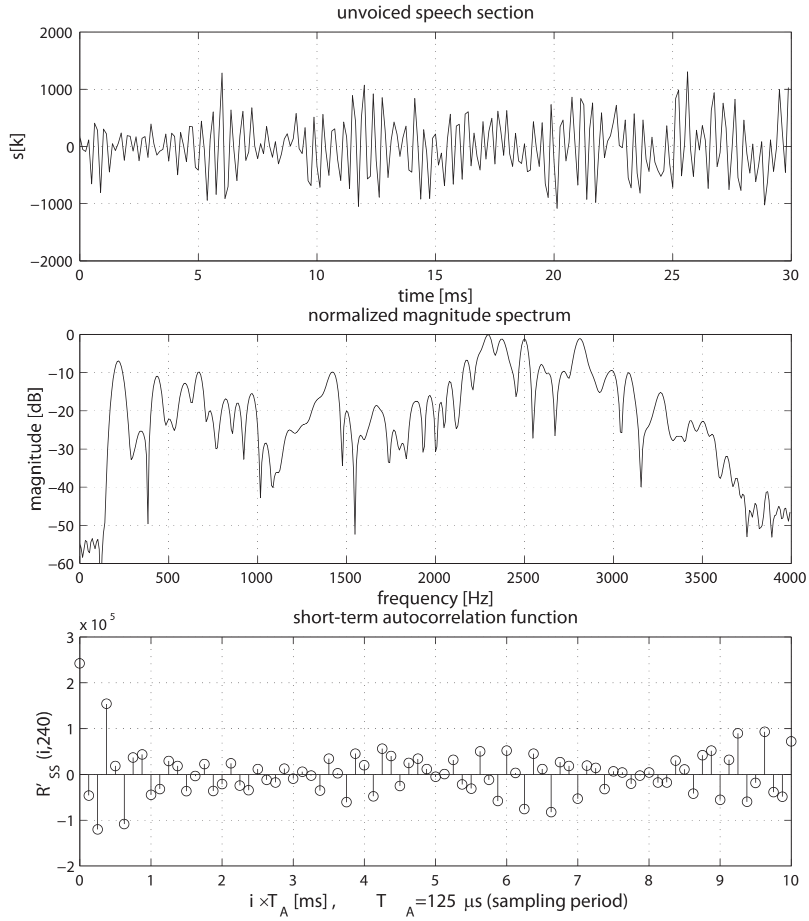

Figure 2.4: Time signal, magnitude frequency spectrum and ACF of an unvoiced speech frame, courtesy of [18].

spectral envelope and with it the formant positions do not change in a discrete manner during speech but have smooth transitions. It was found, however, that for periods of 20 to 30 ms, speech signal sections can be treated as stationary. That means that its properties regarding the power density spectrum can be assumed to be constant for a certain period [18]. Therefore, when analyzing the signal it is possible to concentrate on one frame at a time and describe its properties without worrying about the continuous transition between formant positions. These are suited for extraction as model parameters for transmission. Likewise, on the other side of the line, the decoder would receive the parameters and use them for synthesis so that PCM speech data is reproduced frame by frame.

2.3 Speech Production Model

In order to obtain a basis for parametric coding, the natural process of speech production, called *phonation*, can be expressed by a simple model. Since the human ear is largely insensitive to phase information, it is sufficient for the model to reproduce the frequency characteristics of a given reference speech frame as efficiently as possible. As can be seen in the diagram in Figure 2.5, the central component thereby is a digital filter. It is driven by white noise moderated by a gain value. As analogies to the natural process, the noise source corresponds to the lungs and the filter to the vocal tract: for the filter, the spectrally constant noise serves as the target through which the frequency response of the filter gets expressed according to whichever formants should be produced, [11]. In other words, an approximation of the frequency spectrum of the frame is generated by attenuation of some components through the filter.

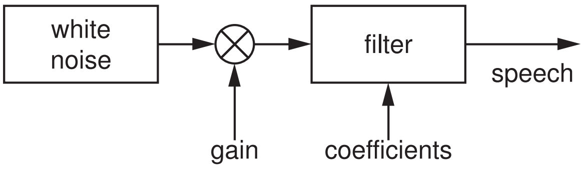

Figure 2.5: Basic speech production model.

The change of sound in speech is reflected by time-varying filter coefficients and gain values, again, held constant for the duration of a single frame. The next sections explain how the presented model can be utilized in an encoder/decoder arrangement to derive the parametric representation of a given frame.

2.4 Linear Predictive Coding

As already mentioned, correlation among the amplitude samples of a speech signal $s[n]$ exists. This observation suggests that an estimation $\hat{s}[n]$ can be derived from N preceding samples $s[n-1], s[n-2], ...$ for the upcoming $s[n]$ [4]. An equation implementing this idea is called predictor [18]. Interpreting the signal as an *autoregressive* process [11] leads to one possible predictor form, that of a linear combination. Its application is called linear predictive coding (LPC).

$$\hat{s}[n] = \sum_{i=1}^{N} a_i s[n-i]$$

The above difference equation can be interpreted as a digital filter with infinite impulse response (IIR) of order N and coefficients a_i , which presents the connection to the base model. In the base model, the gain-moderated noise input to the synthesis filter can be related to the produced speech signal by a system function in z-domain, [24]. With $S(z)$ as the speech and $X(z)$ the noise input, the system function is given by

$$H\left(z\right)=\frac{S(z)}{X(z)}$$

.

According to the model, the purpose of the filter is the approximate reproduction of the frequency spectrum of the desired speech frame. Therefore, the nominator polynomial can be omitted if the denominator order is chosen high enough leading to the all-pole approximation

$$H\left(z\right)=G\frac{1}{A\left(z\right)}$$

whereby

$$A\left(z\right)=1-\sum_{i=1}^{N}a_iz^{-1}$$

with satisfying spectral shaping potential for high enough N, and gain G. Its transformation into time domain yields

$$s\left[n\right]=Gx\left[n\right]+\sum_{i=1}^{N}a_is[n-i]$$

corresponding to the model with the addition of noise to the pulse response of the filter. The similarity to the linear predictor is evident. The objective of the encoder now is to estimate the unknown coefficients a_i pretending that they had produced the speech in a complementary synthesis filter, as in the model. Thereby, it is clear that the random term $Gx[n]$ that originally drives the imaginary synthesis must be thought of as channel noise or some other form of error at the predictor.

The way to the wanted coefficients is via the error of prediction given by

$$\hat{d}[t]=s[n]-\hat{s}[n]=s[n]-\sum_{i=1}^{N}a_is[n-i]$$

$\hat{d}[n]$ is also called the *LPC-residual signal*. Because speech inherently has a zero mean value, its variance, the mean square error (thus identical to the signal power), is given by

$$Var\left(\hat{d}[n]\right)=\mathrm{E}\left\{\left(\hat{d}[n]-0\right)^2\right\}=\mathrm{E}\left\{\hat{d}^2[n]\right\}=\mathrm{E}\left\{\left(s[n]-\sum_{i=1}^{N}a_is[n-i]\right)^2\right\}$$

It serves as measure for the estimation performance and must be minimized. Minimization is by setting to zero its partial derivate with respect to a_i. After switching the positions of summation and mean operators and substitution with the ACF as defined earlier, a linear system of N equations is obtained:

$$\sum_{i=1}^{N} a_i \phi_n(i-j) = \phi_n(j), j = 1, \ldots, N$$

Consequently the encoder must solve the system to obtain his (the estimated) coefficients a_i. In [34] it is shown how this system can be solved with the popular Levinson-Durbin algorithm, which is also applied in the TETRA standard.

At this stage of complexity, the decoder as the synthesizer, is identical to the arrangement in the base model and receives gain and filter coefficients from the encoder. The encoder inverts the operations in the production model. An example for the frequency responses for analysis and synthesis filter can be seen in Figure 2.6.

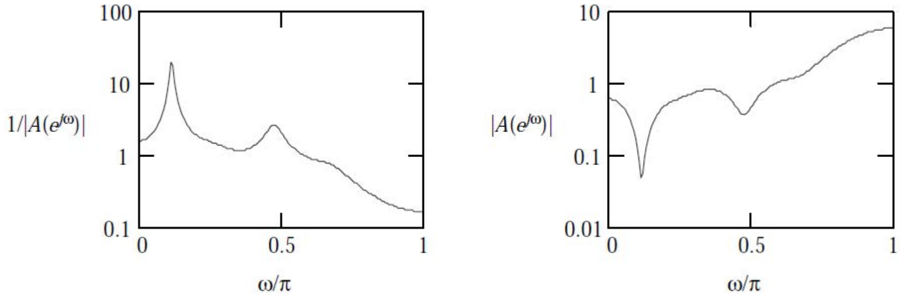

Figure 2.6: Magnitude plots of example analysis and synthesis filters, taken from [11]

The time-domain interpretation of the prediction is that the autocorrelation signal model exploits repetitions of amplitude patterns in its self. Thus it is sufficient to transmit only the changes of the signal in respect to its immediate past. To express the performance of the analysis, the prediction gain PG is defined as the ratio of the variance of residuals and the variance of the input signal, [18, 24].

$$PG = \frac{\sigma_S^2}{\sigma_R^2}$$

Within the bandwidth of 4 kHz, the number of formants for most vocal sounds is four because less significant higher patterns are cut off by the relatively low but sufficient sampling frequency. For their representation through the spectral envelope, at least four conjugated poles are required in the synthesis filter function. It was found that PG saturates with the number of coefficients. Its increase for more than ten coefficients is marginal and in practice ten is the number chosen [24].

2.5 Short-term and Long-term LPC

In the setup described so far, one problem emerges when considering periodicity in voiced speech, as introduced earlier. Upon analysis in the prediction filter, ten coefficients only affect ten past signal samples and thus, can only capture relations in this short range. That is sufficient for the short-term autocorrelation patterns as they appear in both voiced and unvoiced speech portions, as can be seen in figures 2.3 and 2.4 with ACFs depicted. In voiced portions however, strong correlations occur in a range above 43 samples. This number of samples corresponds to the pitch period of the speaker's voice. In order to reflect these relatively long-term correlations, the filter order would need to be expanded to up to 160 coefficients[3]. A number of problems and inefficiencies are the consequence to the adaption of such high filter orders, [18]:

- For unvoiced frames, calculating and transmitting so many coefficients to express correlations that do not exist is a waste of processing power and channel bandwidth.
- For both voiced and unvoiced frames a number of intermediate range coefficients are wasted for mid-range relations that do not occur nor contribute to the quality of the end result.
- Another cause for inefficiency is the dynamic nature of the voice pitch itself, which is 20 to 160 samples. Since the high number of coefficients is required for the longest pitch periods only, more common higher pitch renders parts of the spent encoding process ineffective.

Thus, the filter order shall remain as low as possible so that maximization of the prediction gain and low data volumes are combined. A solution can be illustrated by extending the base model. Obeying the restriction to only ten filter coefficients, there is no way to produce the kind of periodicity found in voice. So attention is directed towards the part of the filter excitation. For unvoiced frames, the model is already sufficient and in this case excitation shall remain unaltered. For voiced sequences however, the source now is switched to an additional function, as depicted in Figure 2.7. In order to insert voice pitch into the system, the filter is excited by an impulse train. It can be imagined as a series of peak amplitudes at certain intervals. The period between pulses is then adjusted to correspond to the desired pitch frequency of the speaker's voice, [11].

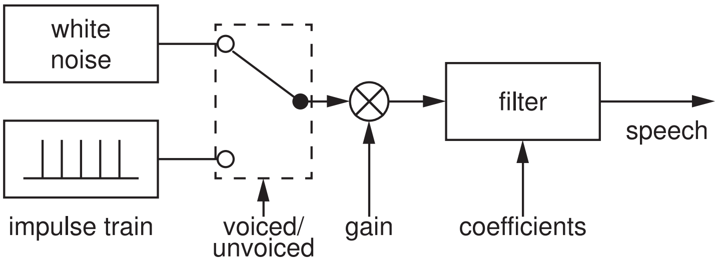

Figure 2.7: Modified speech production model.

Generally, the aim in analysis is to whiten the residuals as far as possible, because that means perfect extraction of spectral characteristics. When compared to the input speech signal, LPC

---

[3] Pitch of 50 Hz.

residuals have their spectral envelope removed for the most part. See Figure 2.8 for an example of LPC residuals.

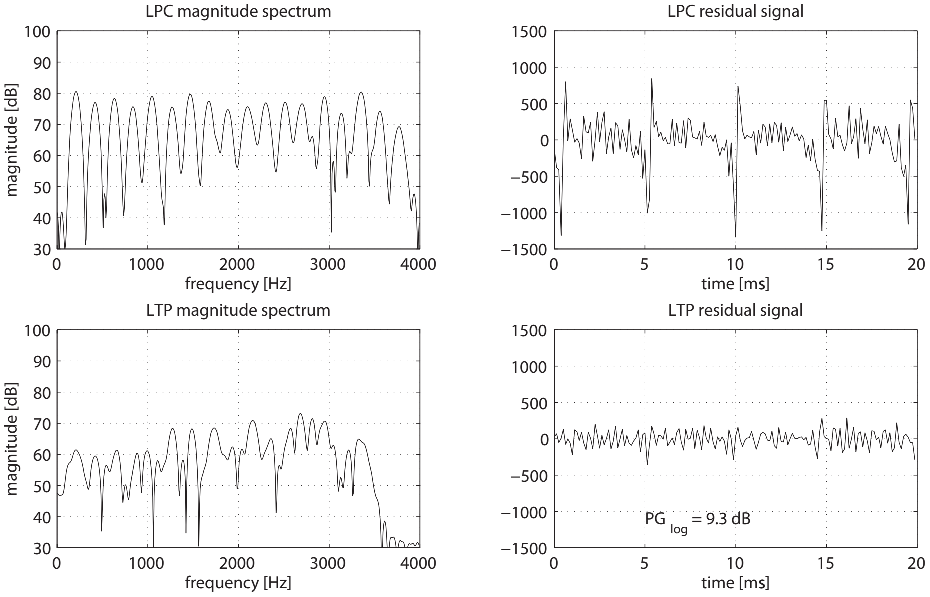

Figure 2.8: Typical LPC and LTP magnitude spectra and residual signals, after short-term (above) and long-term (below) analysis filtering.

Now after the above modification to the model, the LPC-residuals produced by the unmodified prediction filter have changed too. One can see that the periodicity inserted by the impulse train remained untouched by the low order filter. Since the predictor is expected to as closely as possible reverse the procedure in the production model (note: maximize prediction gain), the solution is to add in series a second prediction stage called the long-term predictor (or LTP stage[4]), shown in Figure 2.9. It is entirely dedicated to the retrieval of the pitch information.

Again, the prediction works complementary to the production model. The approach is the same as in the short-term LPC variant [18]:

$$d[n] = \sum_{i=-I}^{I} \check{b}_i \hat{d}[n - L - i]$$

The only difference here is the influence of the desired pitch period L and that the subject of prediction is the LPC residual signal $d[n]$ instead of the original speech input $s[n]$. L can be interpreted as the lag that causes the estimation to be based on a fixed small number of samples $[L - I, L + I]$ in the signal past. L must be set so that b_i can capture the peak caused by the impulse train. With fixed L, the solution is the same as for the short-term predictor, that is, the minimization of power for the related error signal

---

[4] Although being low-order LPC itself.

$$\check{r}[n] = d[n] - \check{d}[n]$$

Minimization is analogous as in LPC by setting its derivate to zero which leads to a system of equations. The desired pitch period L must be found by solving the system for all possible L in a certain range. The one that results in $\check{r}[n]$ with the least power, is selected. The approach with $I = 0$ is called *one-tap* predictor and is widely used for its simplicity, i.e. no system of equations has to be solved. It is then sufficient to maximize

$$\frac{\left(\sum_{n=0}^{N-1} d\,[n] - d[n-\mathrm{L}]\right)^2}{\sum_{n=0}^{N-1} d^2[n-\mathrm{L}]}$$

by calculating powers in an appropriate range for L.

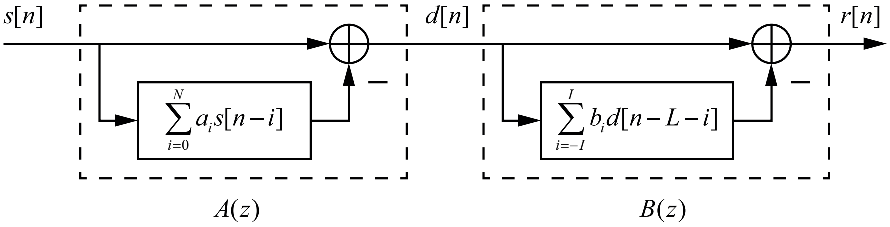

Figure 2.9: LPC and LTP filter cascade

The pitch frequency has less stationary character than the formant characteristics expressed in the LPC coefficients and therefore must be evaluated more than once per frame. To reflect this, the frame is subdivided into a number of *sub-frames*, usually four. Long-term analysis is then conducted for all sub-frames.

Simple LPC Encoder Structure

An overview of how the modified speech production model is applied in an encoder is now given. A possible encoder based on the discrimination of voiced and unvoiced speech is shown in Figure 2.10. It extracts several parameters, amongst them voicing information, on which synthesis in the decoder is based on. Methods for the classification of a frame being of voiced/unvoiced nature should not be treated here, but obviously high power is representative for voiced frames, and high zero crossing rates suggest unvoiced character [8].

The encoder setup is the following. The input signal is partitioned into equally sized frames in which it is assumed to be stationary [18].

1. For each whole frame in formant analysis, LPC coefficients are found that express the spectral characteristic in a short range.
2. Next, the LPC residual signal is generated by analysis filtering through A(z). In the case of voiced frames and hence long-term components, the voice pitch could not be extracted in this step and remains in the residual signal, as in 2.8.

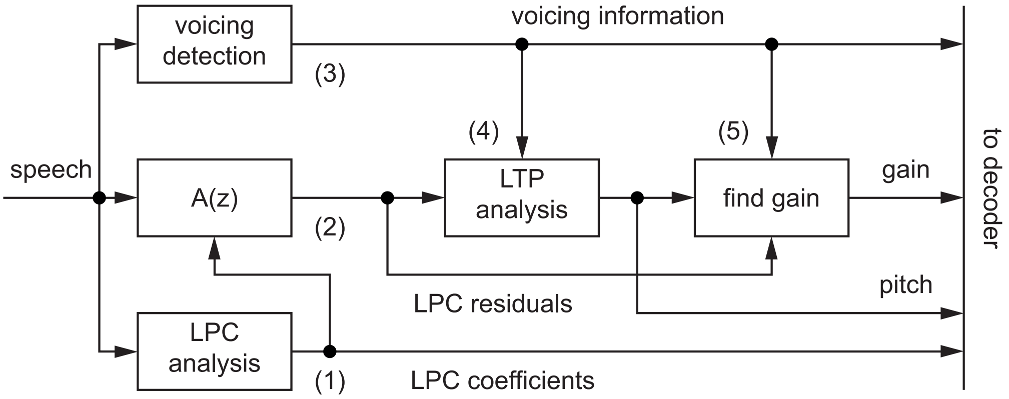

Figure 2.10: Simple coder structure based on the modified speech production model.

3. The frame is then examined whether of voiced or unvoiced nature. The information is used in subsequent steps.

4. If the current frame was classified as voiced, LPC residuals in turn are fed to a second dedicated LPC stage for pitch extraction. This step is, too, carried out on sub-frame basis.

5. Gain calculation is based on LPC residuals and voicing information. This step is carried out on sub-frame basis.

LPC coefficients, voicing information, gain and pitch are sent to the decoder which proceeds as in the production model synthesizing the speech reproduction.

The pure LPC method with its discrimination of voiced and unvoiced frames enables low bit-rate coders (the so-called *vocoders*). The simple model however is the cause for poor quality and the introduction of unnatural sounding artifacts in synthesized speech. The model fails in the following respects [11]: In practice, there are frames that cannot entirely be classified as either voiced or unvoiced. In the employed model, such transitional frames are not accounted for, resulting in audible distortions. Also environmental background noise during a conversation may lead to wrong classification of frames causing unintelligible output. The limitation is that of the use of either white noise or a pure impulse train which is too coarse an abstraction for real speech. While noise leads to reasonable results for unvoiced sounds, voiced ones require a much richer excitation spectrum than an impulse train can offer. Above all, the LPC method is expected to recreate the formants of speech and therefore expects noise as input. If excitation is only by a series of pulses with narrow spectral characteristics, the filter lacks important frequency components. As mentioned, the LPC method does not intend in any way to preserve the time-domain waveform of the signal. The only means of sound representation is through the approximation of its magnitude spectrum per frame. Phase information from the original signal is indeed insignificant for unvoiced sounds but greatly enhances naturalness of voiced components. Today, the discussed encoder design is seen as inadequate for most applications.

To understand the TETRA codec, further developments in coder design must be studied and will be introduced in the following sections.

A closed-loop approach

Obviously, the main disadvantage and cause for poor synthesis in the simple LPC vocoder is the switched excitation source in the decoder. Successive codec designs therefore follow the idea of discarding the discrimination between voiced and unvoiced frames and instead let the encoder itself provide the suited excitation signal directly by transmission.

Even a vague approximate representation of the signal through pitch, gain and LPC coefficients allows its exact reproduction if the residuals are transmitted unaltered to the decoder. In other words, the LPC residuals present the ideal excitation signal for the decoder with any set of determined coefficients. This is due to the reciprocal nature of analysis and synthesis filters (refer to Figure 2.6 with inverted system functions). However, parameters as well as residuals cannot be transmitted as they are which would render all intentions for data volume reduction pointless. The crucial instrument to enable the transmission of the results therefore is that of careful quantization. The concept has already been introduced at the beginning of this chapter. For now, without suggesting further details it is sufficient to think of quantization effects as additive white noise to its target signal, thereby representing the quantization error. Details on quantization techniques are treated in the next section. The coder arrangement with abstract quantization and transmission of residuals is depicted in Figure 2.11.

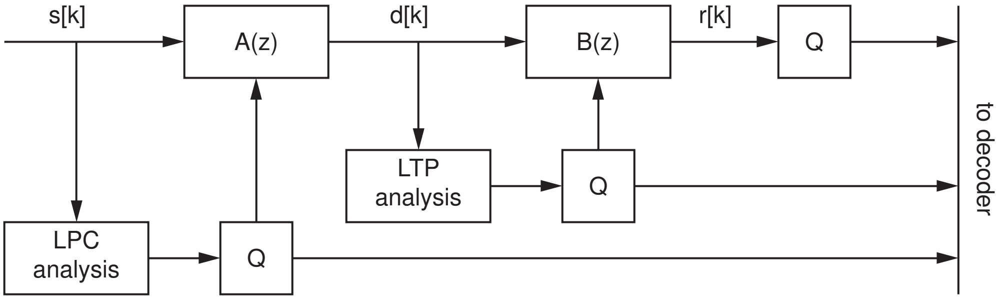

Figure 2.11: Coder structure with quantization of parameters and residual signal

In the structure, the voicing part is gone. Instead, all parameters are transmitted to the decoder in quantized form. The main difference is the transmission of the LTP residual signal[5]. In the diagram, gain calculation is omitted. Gain can be thought of as being included in the residual signal.

Essentially, the whole procedure is aimed at bringing the residuals into a form that is suited for quantization and allows its efficient transmission together with LPC and pitch information. The final residual signal is thus the result of a two-stage process that minimizes its variance (power). The reduction in variance means less dynamic, i.e. a smaller amplitude range. With the same number of bits, quantization intervals can be laid out much tighter resulting in higher SNR in residual representation.

Notice, that no matter how precise the prediction is, the final result at the decoder output is independent from PG. Ignoring effects of possible error introducing representations of co-

---

[5] Due to the employment of quantization to the residuals, LPC-based encoders often are classified as hybrid encoder types, incorporating both parametric elements as well as waveform approximating ones.

efficients and gain, it is only dependent on the residual quantization error disrupting the ideal excitation signal. In a system as it is depicted in Figure 2.11, the purpose of analysis therefore is merely that of variance reduction. Apparently, this is because the synthesis in the decoder is based on different data than the analysis in the encoder. That is because analysis filters have non-quantized input and the effects of quantization on the final decoder output are neglected in the encoder.

Speech quality however can be enhanced by eliminating this shortcoming and making the overall SNR dependent on PG. How the effects of quantization can be incorporated into the prediction decision will be discussed in the next section. The idea of considering the quantization error during the encoding process is central to the final coder design. Its structure is best understood through details on the applied quantization methods to which the next section is dedicated.

2.6 Vector Quantization

According to Figure 2.11, quantization is the elemental tool for data volume reduction and is integral to the here studied coder design. Scalar quantization methods as applied to isolated samples were already mentioned at the beginning of this chapter and are generalized in the following.

Basically, vector quantization (VQ) is the substitution $q(\mathbf{x}) = \mathbf{y}_I$ of an input vector $\mathbf{x} = [x_1, x_2, ..., x_N]^T$ with a sufficiently similar one $\mathbf{y}_I$ from a set of available vectors Y of the same dimension N. The set of available vectors $Y = \mathbf{y}_1, ..., \mathbf{y}_M$ that are considered for substitution is called the codebook of size $\log_2 M$ bits. Selection of $\mathbf{y}_I$ is by minimization of a given distance function $d(\mathbf{x}, \mathbf{y})$ between the input vector and each vector of the codebook. A common distance function is the squared Euclidian distance [11]

$$d\left(\mathbf{x}, \mathbf{y}\right) = \parallel \mathbf{x} - \mathbf{y} \|^2 = \sum_{i=1}^{N} \left(x_i - y_i\right)^2 .$$

Wherever Y is known, the input vector $\mathbf{x}$ can then be represented by the index I to its codebook vector minimizing $d(\mathbf{x}, \mathbf{y})$. In the context of an encoder/decoder arrangement, this means a great reduction of data to be transferred to the decoder. In practice, and as will be seen in the final coder design, VQ is applied to almost all resulting model parameters produced by the encoder. For example, in the variant from Figure 2.11, LPC coefficients (ten elements) and residuals (sub-frame width) might be vector quantized, pitch and gain on the other hand, being scalars, can be transmitted directly. Note that the above distance function is well applicable for signal samples, but inappropriate for targets such as the LPC coefficients whose quantization will be treated in a later section.

The essential parameters for the application of VQ are the number of bits $\log_2 M$ for code vector enumeration and the code vector dimension N. As an example, the quantization of the residual signal should be considered [18]. Assuming four sub-frames with a width of $N = 40$ and a sub-frame time of 5 ms and choosing a codebook size of $2^M = 2^{20}$, accumulating transmission data is already relatively low at 4 kb/s. However, one problem thereby is the required memory amount for codebook storage. With 16-bit samples, each of the codebook vectors y_j the input speech frame is compared against requires $2N = 80$ bytes, totaling at $80 \cdot 2^{20}$ bytes, which is well above 80 MB. Another issue with this form of codebook is the complexity arising when searching it for the best match. Both, memory and computational requirements exceed the capacities in typical applications by magnitudes. Thus, means for the reduction of codebook sizes are necessary.

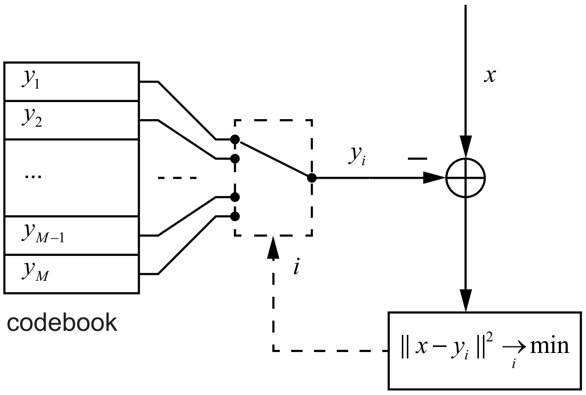

Figure 2.12: Vector Quantization, as in [18].

The advantage of VQ with a high number of elements per vector is that it can exploit linear as well as nonlinear dependencies between vector components that are otherwise hard to express. As logarithmic quantization tries to assign larger errors to less common amplitude values (see beginning of the chapter), so does VQ accept the larger errors for rare sequence patterns of amplitude values. It outperforms scalar quantization because there, bits are wasted to the relatively precise representation in an insignificant scope. This holds even for uncorrelated signals [18].

Split VQ

One way to increase VQ efficiency is by splitting the input vector into more than one part of not necessarily equal length [24]. Consequently, each sub-vector is then quantized by use of a separate codebook. The advantage thereby is that different parts of the vector can be assigned with different grades of quantization precision, i.e. codebook sizes in bits, according to requirements. Actually, the result is that of having a number of independent parameters each with its own VQ stage.

Gain-Shape VQ

An essential approach for the reduction of complexity is *gain shape* VQ (GSVQ) [18, 24]. It is especially interesting because of its influence in the final coder structure that will be described in Section 2.8. The discussion of the LPC coder in Figure 2.11 revealed that the potential of quantization is significantly affected by the variance of the input signal. As observed in the context of scalar quantization, the situation is the same with VQ. The idea behind gain shape VQ is to exploit the fact that various input vectors might share the same or similar sequence of values (shape) although at different power levels (variance, here: gain). The codebook is therefore split into two separate parts, one for shapes and one for gain levels. In order to eliminate the influence of variance, the input vector **x** is normalized and compared against normalized target vectors from the dedicated shape codebook. The gain of **x** which is given by

$$\sigma_x = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$$

is quantized as a scalar by means of the gain codebook. The output of the quantization then is the indices to the best shape approximation combined with the optimal gain level. The testing of each gain level in combination with each shape entry can be quite complex for large codebooks. As a consequence in practice, both parameters are determined in two distinct steps.

1. In the first step, the non-quantized gain is used to determine the best shape vector.
2. In the second step, the best shape vector selected in step 1 is used to search for the optimal entry in the gain codebook.

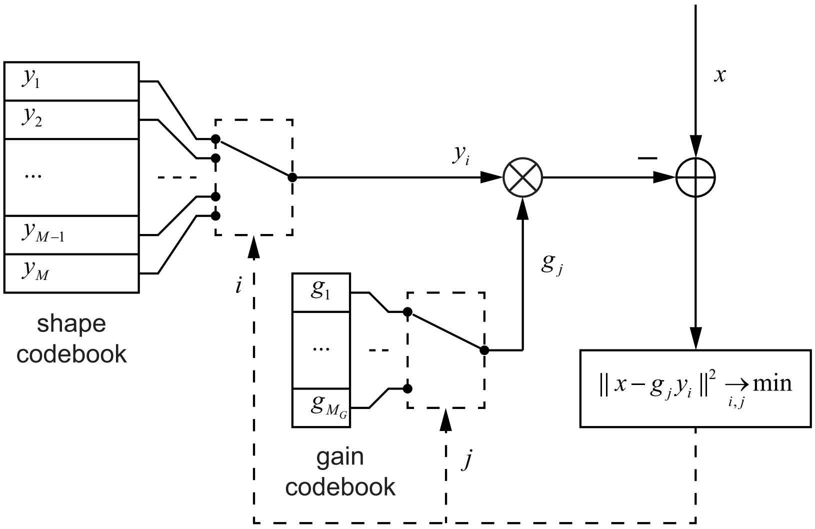

Figure 2.13: Gain-Shape Vector Quantization as in [18].

Compared to the regular VQ approach codebook size is reduced greatly. For illustration, GSVQ should be applied to the previous example with the 20-bit codebook and a vector dimension of $N = 40$. Let the 20 bits arbitrarily be divided into 12 bit for the shape codebook and an 8-bit gain quantization. Again, with 2 bytes for each of the 40 vector elements, the required storage space for both codebooks totals at $2 \cdot 40 \cdot 2^{12} + 2^8$ which is about 328 kB.

2.7 Analysis by Synthesis

Both coder variants discussed so far are open-loop designs, that means parameters are determined once, then quantized and sent to the decoder. For low bit-rate systems, it is essential to optimize synthesis quality. Modern designs therefore adopt the far more potent closed-loop approach. In Figure 2.14, the general concept is depicted.

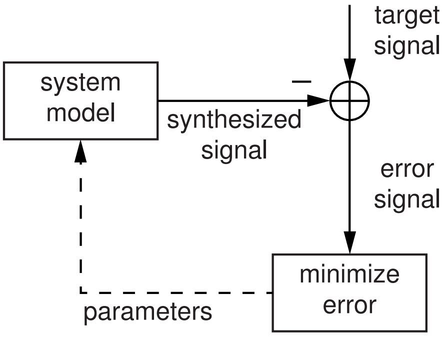

Figure 2.14: Closed loop

According to the given model, the input signal is to be reproduced already in the encoder. Since an analytical determination of the optimal parameters might not be feasible, model parameters are applied in a trial and error fashion. The synthesized signal is then compared to the original for each combination of model parameters. The resulting difference is an expression of how close the chosen parameters match. The fact that synthesis takes place in the encoder gives the method of analysis-by-synthesis (AbS) its name. It was first applied in an LPC speech coder in [5].

The drawback of the previously described coder design is that the prediction gain has no influence on the quality of the decoder output. By introducing the AbS principle to LPC, this disadvantage is eliminated. For an analytical step-by-step derivation of the structure see [18]. For quick illustration, the closed-loop LPC design is best derived from the previous variant (the one in Figure 2.11) by replacing the abstract quantization block with a VQ stage. The motivation for VQ has already been given in the previous section.

Notice that VQ as depicted in Figure 2.12 is in itself a closed-loop system with the codebook corresponding to a reduced system model and the index to its parameters. When replacing the quantization stage in Figure 2.11 with that from Figure 2.12 in the open-loop coder, the excitation signal resulting from analysis filtering is then the input to the closed-loop VQ. In order to apply the principle entirely on encoder level, the structure must be just slightly rearranged. This is by setting the speech signal as the reference signal of the closed-loop. The result is shown in Figure 2.15.

Now, the source of the excitation signal is no longer the residuals from LPC analysis. Instead, in a search loop all possible codebook vectors are fed to a subsequent synthesis stage. It consists of the pitch and LPC synthesis filters and basically can be thought of as an embedded decoder. The difference between the synthesized speech and the original serves as the error measure. As in VQ, the excitation sequence (of sub-frame length) leading to the least error is

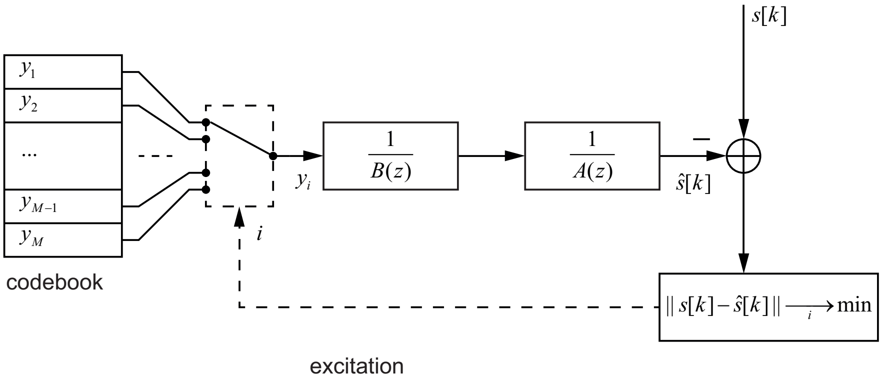

Figure 2.15: AbS LPC coder

selected and its index sent to the actual decoder. Essentially, the former calculation and quantization of residuals is substituted with the generation of a signal with the same purpose. Note that this illustration does not consider steps such as determination of pitch and LPC coefficients that are required to drive the synthesis part.

Filter memory handling

Before the full encoder structure is given, a detail about synthesis filter memory management during the codebook search should be pointed out. Note that recursive digital filters, besides their current coefficients also retain a state from the previous output calculation. In a tenth-order filter, these memories or states consist of the last ten output values. When switching on a filter, i.e. feeding a step function, the first ten+1 output values are called its *transient response*. After having calculated ten+1 values the filter has reached its *steady state*. After the input is switched off the remaining ten+1 output values are called *decay response*. The transient response is based on a zero-filter state since no previous calculations took place. In the above arrangement, at time k, filter memories are the result of the excitation of the tested code-book entry at time k-1. If not explicitly managed, the resulting memory from each search step influences the next calculation and the results are distorted. Therefore, after each codebook vector, the filter memories must be reset to the state they were in when the search began. In practice, there is a more efficient solution: due to the linearity of the system, the decay response of the filter cascade can easily be found and subtracted from the input signal just once before the search commences. During the search the filter cascade is initialized with zero-states for each codebook vector, thus ensuring proper search results and less handling of filter memories [18]. Figure 2.16 illustrates this additional step.

2.8 Code Excited Linear Prediction

The first system combining AbS and VQ was proposed in 1985 [48] and called *Code Excited Linear Prediction* (CELP). Its name suggests what has been described in the last section: the

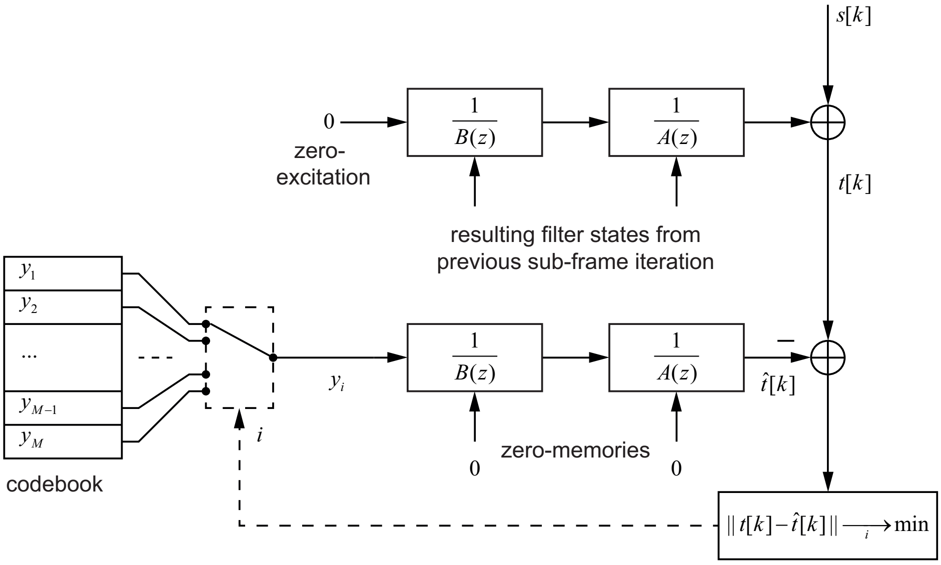

Figure 2.16: AbS LPC coder with decay response subtraction

generation of the synthesis filter's excitation signal from codebook entries. Many of today's medium and low bit-rate speech codecs are based on this design.

Perceptual Weighting

Because the synthesized speech signal produced in the AbS search loop barely resembles the original time-domain waveform, the mean square error measure performs less than adequate. To counter this effect, in the CELP coder, a property of the human auditory system, the *masking effect*, is exploited to further enhance synthesis quality. The presence of a single tone in the frequency spectrum tends to cancel the audibility of other tones in its spectral vicinity, if they are of sufficiently low magnitude. The higher the magnitude of a tone, the higher the degree and range of its suppressive effect. Each tone therefore is accompanied by a masking curve as seen in Figure 2.17. Considering the spectral envelope of a voice frame, this implies that regions of high power (formants) have a higher potential of canceling out weak components than regions of low power do. Thereby, the resulting masking curve resembles an attenuated version of the actual envelope, [18, 24].

The application of masking in speech coding is called *perceptual weighting*. During the AbS search loop, the error signal power is used to determine the best codebook vector. The idea now is to de-emphasize errors in more tolerant frequency regions, i.e. those of the formants. Since the LPC coefficients already express those, the error weighting can be achieved by a weighting filter given by

$$W(z) = \frac{A(z)}{A(z/\gamma)}$$

with γ usually in $[0.8, 0.9]$.

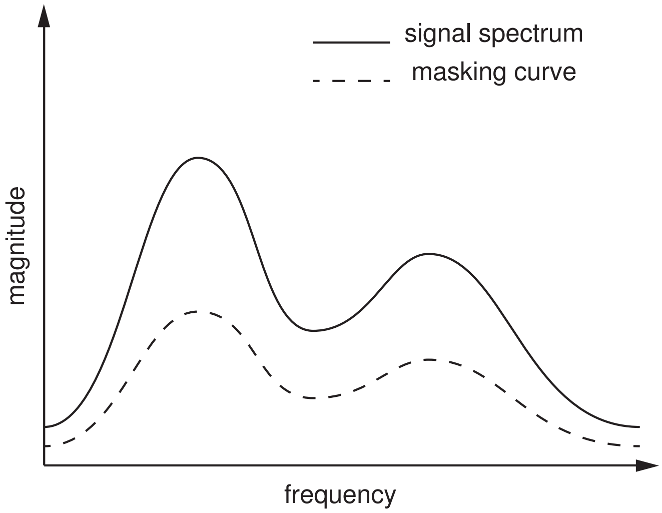

Figure 2.17: Example of signal spectrum and according masking curve, traced from [11]

For the same reasons as with the subtraction of the decay response in filter memory handling, it is more efficient to apply W(z) to the speech signal in the first place. Thereby, $W(z)$ is integrated into the synthesis filter so that $\frac{1}{A(z)}\frac{A(z)}{A(z/\gamma)}$ is reduced to $\frac{1}{A(z/\gamma)}$.

The eventual CELP structure follows the one in Figure 2.16 with the addition of conceptual weighting and other changes. The main difference is that in the CELP coder the pitch prediction is integrated into the closed-loop thereby eliminating $\frac{1}{B(z)}$ from the synthesis cascade. The excitation signal source thus becomes more complex than a single codebook which makes the AbS loop more complex.

Encoding Procedure

The whole procedure is described in the following [18]. There are two phases:

- whole frame processing
- sub-frame processing

The first phase includes the LPC analysis based on the whole frame in the same way as explained in the simple coder designs. The obtained coefficients are used for all subsequent applications of $A(z)$ until the next frame. They are quantized (see Section 2.9 for details) and ready for transmission to the decoder. After this, non-quantized and quantized LPC coefficients are available. Finally, the original speech signal is weighted through $W(z)$.

The first step in sub-frame processing is the subtraction of the impulse response of $1/(A(z/\gamma))$ from the weighted speech signal, as discussed above. The result serves as the reference signal for the subsequent AbS search loop that should find the optimal excitation signal.

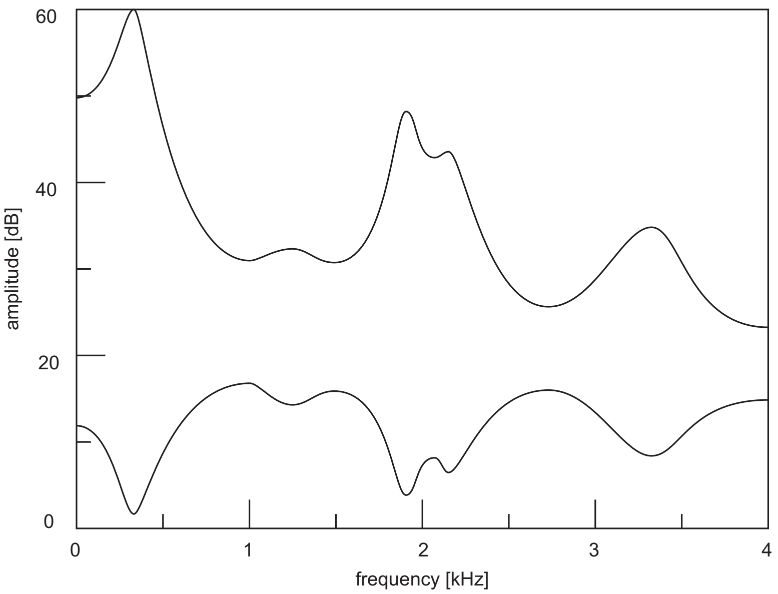

Figure 2.18: An example of the spectral envelope of speech and the weighting filter frequency response, traced from [3]

The excitation signal is generated from two different codebooks, the stochastic and the adaptive one, see 2.20. The stochastic codebook is called that way because it is defined in a training procedure beforehand, as described in the section about VQ. In a GSVQ manner, both codebook contributions are moderated with a gain value each. To form the candidate excitation signal, both results are added together in a certain proportion determined by the gain mix. The procedure is as follows:

- calculate adaptive codebook contribution, obtaining pitch
- calculate stochastic codebook contribution and gains, obtaining index
- quantize gains for each contribution, obtaining gains-index

The adaptive codebook is not a codebook in the sense of VQ but rather a sequential first in-first out (FIFO) sample buffer in which the samples of previously chosen excitation signals (on sub-frame level) are stored. It is called adaptive because it is modified over session time. More precisely, after each sub-frame processing the resulting excitation is inserted into the buffer. In the decoder this adaptive codebook can easily be maintained because exactly the same excitation signal is reproduced from the received parameters. The pitch is determined by finding a vector $\mathbf{b}_l$ of sub-frame length somewhere in the buffer where l, the lag, denotes the buffer offset from its beginning.

$\mathbf{b}_l$ is found in a full search for the whole range[6] of l which is $l = 20, ..., 147$. For each value of l, $\mathbf{b}_l$ is fed to the synthesis filter $1/(A(z/\gamma))$ and the output is subtracted from the reference

---

[6] Again corresponding to the voice pitch range.

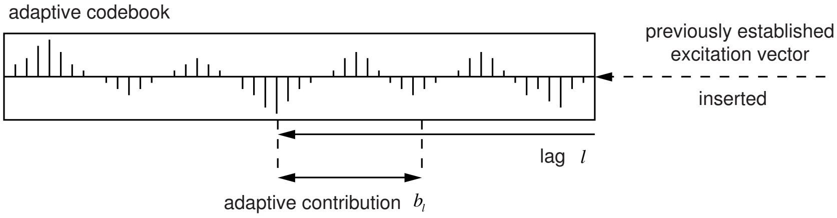
adaptive codebook
previously established excitation vector
inserted
lag l
adaptive contribution b_l

Figure 2.19: Adaptive codebook as depicted in [18]

signal. Thereby $\mathbf{b}_l$ is tested as excitation signal and the l leading to the error signal with the lowest power is selected as pitch L. As an implementation of GSVQ, an optimal gain value β is determined that further scales the *shape* of $\mathbf{b}_L$ to the reference signal.

One note about the purpose and effect of adaptive codebook for voice components: The target excitation signal's waveform is approximated by an adequate section taken from past excitations. This yields good results especially for voice-dominated frames, because pitch is introduced into the signal by involving the previous excitation signal's peaks .

The contribution $\mathbf{b}_L$ from the adaptive codebook is just the basis for the desired excitation and must still be refined. In the next step, the deviation from the reference signal is further reduced by adding to $\mathbf{b}_L$ an appropriately scaled vector $\alpha\mathbf{c}_i$, α being the applied gain, from the stochastic codebook. $\alpha\mathbf{c}_I$, is also called the innovative contribution because without it the excitation signal would be entirely based on content from its own past.

Both gain values β and α used so far were optimal for the shapes $\mathbf{b}_L$ and $\mathbf{c}_I$ they scale. They too are to be quantized according to the concept of GSVQ. Dependencies between β and α can be exploited through the employment of a trained gain codebook containing 2-dimensional vectors each with one component for β and one for α. Figure 2.20 depicts the entire excitation source.

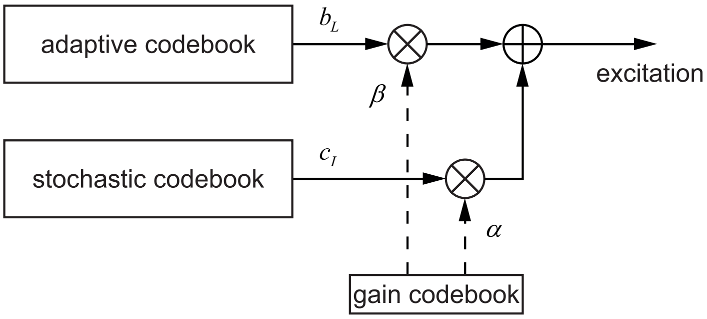

Figure 2.20: CELP codebooks and gain mix.

The processing of the sub-frame is then completed and the parameters that define the optimal excitation are found: the pitch L, the innovative codebook index I, and the gains-codebook index I_G. For the remaining sub-frames, the synthesis filter memory must be updated. This is accomplished by filtering the final excitation signal as the decoder would derive it from the received parameters. As the last step, the adaptive codebook is updated by feeding the selected excitation signal into the FIFO buffer shifting its content to the rear by the sub-frame width.

Algebraic codebooks

For the desired speech quality, the size of the stochastic codebook is crucial. With the common sub-frame width of 40, a codebook size of between 12 to 40 bits is required to achieve satisfying results. Searching and storing up to 2^{40} vectors however is impracticable [18]. Therefore alternative codebook structures that allow for more efficient processing were introduced. One such structure is that of algebraic codebooks in algebraic CELP (ACELP). What makes a codebook *algebraic* is that it is not stored in memory but instead, relies on a set of rules allowing the generation of its vectors directly from their associated code representation [7]. The idea is to use sparsely occupied impulse sequences of sub-frame length as codebook vectors, much like impulse trains mentioned in Section 2.5. Such a vector consists of only a small number (usually four) of signed non-zero pulses of unit magnitude. Each of the pulses can be set at predefined positions evenly distributed over the entire vector.

For example with a sub-frame width of 40, the first pulse can be set at position $0, 5, ..., 30, 35$ the second pulse at $2, 7, ..., 32, 37$ and so on. Thus in this example, each pulse requires only three bit plus the sign bit for its encoding.

The advantage of this approach is that the codebook needs not to be stored, but also that it can be searched most efficiently. The search can be conducted so that it is not required to calculate the contribution of each entire pulse combination. The difference between two combinations that vary only in the position of a single pulse can be expressed (as will be seen in Section 5.5). This is exploited by iteratively testing all possibilities by changing only the fourth pulse for a given combination.

2.9 Specific characteristics of the TETRA codec

The structure of a CELP coder has been discussed in this chapter so far. Still, some details not necessary for the understanding of the encoding procedure or specific to the TETRA specification were omitted. Those relevant for the actual implementation should be revealed in the following. This inludes the way how LPC coefficients are used, the method of finding the proper pitch period, as well as the ACELP implementation.

Quantization and interpolation of LPC coefficients

For synthesis, the decoder relies on the LPC coefficients which the encoder continuously provides together with information on the excitation signal. In their unprocessed form, as they are determined in the Levinson-Durbin algorithm, the coefficients require too much bandwidth for efficient transmission. In the TETRA coder without further data reduction, the ten coefficients a_i (each a 16 bit word) would cause 160 bit to be relayed each frame of 30 milliseconds (8 kb/s) [18]. Needless to say, the required quantization results in deviation from the original coefficients and lowers PG. The problem with the LPC coefficients is that the effect of introduced quantization errors cannot easily be measured nor predicted. As a solution to this, alternative

LPC representations that behave more robustly concerning quantization are used. The TETRA specification employs the common Line Spectral Pairs (LSP) transformation, also referred to as Line Spectral Frequencies (LSF) in the literature [23].

Thereby, the predictor expression

$$A(z) = 1 - \sum_{i=1}^{N} a_i z^{-1}$$

is arbitrarily decomposed into two polynomials Q_1 and Q_2,

$$Q_1(z) = A(z) + z^{-(N+1)} A\left(z^{-1}\right)$$

$$Q_2(z) = A(z) - z^{-(N+1)} A\left(z^{-1}\right)$$

of increased order so that

$$A(z) = \frac{1}{2}\left(Q_1(z) + Q_2(z)\right)$$

All $2(N+1)$ roots of Q_1 and Q_2 lie on the unit circle in alternating order [51]. Each polynomial has always one known root at 0 and π respectively, which can thus be ignored. Since the remaining roots are conjugate complex it is sufficient to consider those in the upper half of the unit circle[7]. The actual $N = 10$ line spectral frequencies, ω_i, are the angular positions $\omega_1, \omega_3, ..., \omega_9$ of the polynomial roots of Q_1 and $\omega_2, \omega_4, ..., \omega_{10}$ of Q_2 respectively, all between 0 and π. The advantage of LSF is that quantization errors have only local effects and that due to their spectral representation can easily be interpolated (see below). Moreover, speech sample analysis has revealed statistical relations among neighboring LSF values, which suggests the employment of VQ. In the TETRA coder, split VQ is applied to the ten frequencies which are split into three groups sized 3, 3 and 4 respectively. The first group is quantized with 8 and the other groups with 9 bits each [22]. Thus, per-frame data dedicated to LPC only accounts for 26 bits. Additionally, the error assignment in the according VQ codebook training provides [18]:

- lesser sensitivity to higher frequencies in the human ear
- emphasis on significant formant positions

Each whole-frame, LSF values are derived from LPC coefficients. For sub-frame processing, LSFs are interpolated so that in each sub-frame a linear mix of the newly calculated LSFs and those from the previous whole-frame are used. The 4th sub-frame relies entirely on the recent values.

---

[7] [11] dedicates a whole chapter to LSF

Pitch prediction with fractional delays

As already said, the aim of pitch prediction in a CELP coder is to find the lag and gain for the adaptive contribution. In the implementation with the adaptive codebook, this vector, $\mathbf{b}_L$, corresponds to a section taken from a buffer of previous excitation samples. To find the lag L, portions of the excitation buffer must be compared to the desired target signal (see below). In voiced frames however the actual pitch period often is not an integer multiple of the inherent sampling period. That significantly reduces the effectiveness of the method.

For the desired quality, sub-sample precision of L is required. Higher order LTP filters (with $I > 0$ as given in Section 2.5) have an interpolating effect and provide a higher resolution for L. The disadvantage however is that more $(2I + 1)$ coefficients from long-term LPC must be encoded for transmission, and that the solution becomes more complex. The solution here is to introduce *fractional delays*. That means up-sampling the signals instead of increasing the predictor order. Then, sub-sample precision in combination with the advantages of a single coefficient is given. The interpolation increases the number of samples and thereby the search complexity. Therefore, fractional search cannot be applied on the whole range of L, [18, 24]. In the TETRA coder, fractional delays with a resolution of $\frac{1}{3}$ are used. For the first subframe, fractional search is conducted for L_1 in the range[8] $\left[19 - \frac{1}{3}, 84 + \frac{2}{3}\right]$. For the other three subframes, only the vicinity around L_1, that is $\left[L_1 - 5 - \frac{2}{3}, L_1 + 4 + \frac{2}{3}\right]$, is interpolated for the search [22].

In order to further reduce the complexity caused by the fractional lag search, an open-loop preliminary search stage is introduced. It provides L' and L'_{min} and L'_{max} delimiting the closed-loop search for L_1. This stage operates on the whole-frame level and works essentially in the same way as the single tap predictor described in the introduction to pitch filtering (Section 2.5). The difference here is that the calculation is based on the weighted speech signal (instead of the LPC residual). Also the normalized correlation that is to be maximized is expressed differently [22]:

$$C_k = \sum_{j=0}^{120} s_w\,(2j)\,s_w(2j - k)$$

Algebraic Codebook Implementation

The ACELP implementation in the TETRA codec follows the approach in [7], depicted in Figure 2.21. Thereby, the innovative codebook is arranged as a general framework consisting of two parts. On the one hand, the set of algebraically available sparse code vectors A_k and, on the other, an arbitrarily chosen matrix F that further enhances the selected vector[9] A_k and produces the actual innovative code vector C_k.

The decoupling of codebook and shaping matrix bears several advantages:

- The search procedure benefits from the simplicity of A_k and is not concerned with the final shape of the output vector containing unnecessary components.

---

[8] Perceptually, lower pitch frequencies are less sensitive to prediction errors.

[9] The actual structure of A_k is given in Section 5.5.

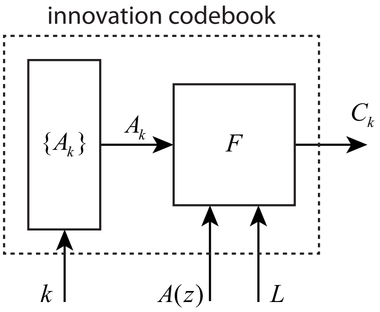

Figure 2.21: Framework for innovation codebook, based on [7].

- Desired speech specific properties of the codebook can be independently expressed in F and won't interfere with the search.

The matrix enhances the excitation vectors in the frequency domain concentrating energies in the significant frequency bands, namely those of the formants [29]. It is a function of the LPC model $A(z)$ and lag L. F is calculated each subframe during the stage of algebraic codebook contribution.

CHAPTER 3

GPU Properties

This chapter treats GPU properties such as architecture and programming model and assumes the knowledge of the old pipeline model as used in computer graphics.

Three main characteristics of applications that can successfully be mapped to GPUs are [44]:

- High proportion of computational activity. GPUs originate from a fixed pipeline model and are not designed to handle programs that strongly rely on conditional evaluations.
- Exploitation of data-parallelism. Many non-graphics related problems can be separated into independent work packages, just as vertices and fragments. Examples are voxels in fluid dynamics, or cells of matrices in matrix multiplications.
- Throughput over Latency. GPUs are built for high overall throughput rather than short response times. In real-time graphics, time constraints are based on the video frame rate. The focus therefore lies on bringing a whole result (an image) within the given time rather than optimizing the performance of individual operations. Many compute-intensive tasks share this characteristic.

When hardware was still not programmable, non-graphics problems were tried to be mapped to depth buffer and color blending operations since those were the only configurable portions on the fixed pipeline [45]. GPGPU methods progressed with the introduction of the shader units [58]. In this stage during the evolution of the GPU, additional programmability was not added with general purpose computing in mind, but to allow for new custom visual effects [1]. The graphics APIs (Direct3D, OpenGL) still had to be learned in order to benefit from the underlying power, to many an unattractive complication [58].

In less than a decade, GPUs evolved from hardwired to highly programmable devices. Stages in the fixed functionality pipeline became gradually customizable. The vertex processing and the fragment processing stage became generic modules, *shaders*. Over the years, vertex and fragment shader units gradually assimilated since programming requirements raised and generality of hardware increased [33]. Eventually, they were unified resulting in the Shader Model

4.0, [44]. The GT200 family of NVIDIA devices used in this work feature this unified shader model.

In [16], a taxonomy of computer architectures is proposed which is still used frequently today. One architecture type is that of *single instruction multiple data* (SIMD). It is often mentioned in the context of parallel computing. As opposed to classical CPU architecture (*single instruction single data*), it denotes an architecture's capability of manipulating multiple operands (a vector) of the same operation in parallel.

3.1 Architectural Overview

Although unified and not resembling the order of classical pipeline stages, the architecture features specialized units dedicated to graphics tasks on different levels (see discussion on GPU unification). The main building blocks in an NVIDIA GT200 device are the streaming processor array (SPA) that does all the customized processing, the scalable memory system and the network for interconnection of the latter. Communication with the host computer is via the host interface. It provides memory access, receives commands and responds to the host. The modules immediately below the host interface in the diagram (Figure 3.1) represent three different paths and supply the SPA with work. The input assembler collects commands related to vertex or geometry (DX10) processing and work is distributed to the SPA by the Vertex work distribution unit accordingly. The unit designated with clip/setup/raster/culling implements all functionality that used to be applied between the vertex and fragment stages in the graphics pipeline (viewport transformations, clipping, rasterization and z culling). After pixel data is setup, the pixel work distribution unit assigns the SPA with the execution of the pixel shader program. Distribution of pixel work is dependent on the pixel location (see ROPs). The compute work distribution assigns general purpose compute kernels, not necessarily related to graphics, to the processor array.

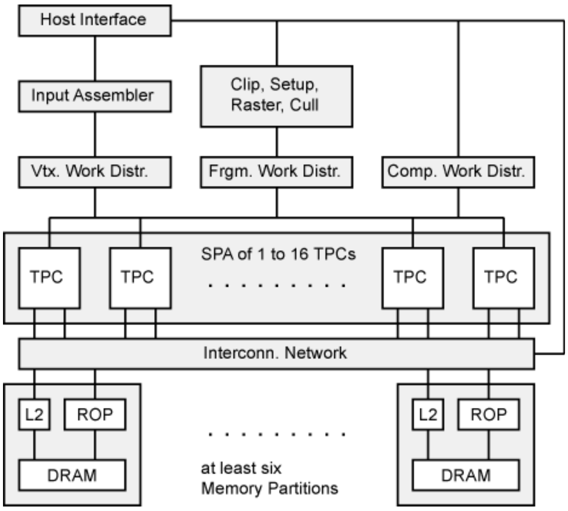

Figure 3.1: GT200 Architecture, based on [33, 40].

3.2 Streaming Multiprocessors

The central structure of GT200 devices is the SPA, a number of identical packages containing the actual processing units. These so called Texture/Processor Clusters (TPC) introduce a further level in the hierarchy of modules in the architecture. The TPC concept not only provides a means for better distribution of resources but also a unit by which the complexity of GPU designs for different performance classes gets scaled by the manufacturer. High end devices for example contain up to 15 TPC while low end cards may contain only one. The hierarchy of the device is illustrated in Figure 3.2. One TPC contains a geometry controller, an SMC (Streaming Multiprocessor Controller), two SM (Streaming Multiprocessor) and a texture unit. The geometry controller handles DX10 specific geometry shader operations and should not be treated in detail here. SMs are multiprocessors executing shader and compute kernel programs. Each holds, beside an instruction and constant memory cache, eight streaming processor (SP) cores and two special function units (SFU). The SPs act like integer and single precision float ALUs in the SM.

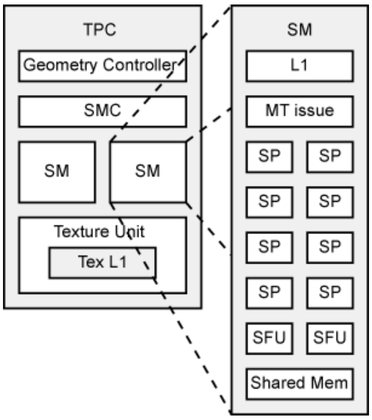

Figure 3.2: Streaming multiprocessors in TPC.

On the lowest level, the SM operates in SIMD fashion, concurrently executing one instruction on up to eight different operands. There are however a number of differences and extensions to the classical SIMD concept. Shader programs and compute kernels are interpreted as single threaded programs each calculating the result to a small arithmetic job that has been extracted from an overall problem by the programmer. Therefore, since shaders were introduced, GPUs sometimes are classified as SPMD (Single Program, Multiple Data) [44].

Unlike classical SIMD architectures, GPUs do not expose vector width and thread handling on the software level [40]. The SPs use scalar instructions manipulating single operands at a time just as a program on a regular desktop CPU [33]. In the case of shader programs, the isolated job is the calculation of vertex attributes or the final color of a pixel. A host program would instantiate a large number of these threads to maximize exploitation of data parallelism. The GT200 architecture with its SMs is designed to relieve host software from the hereby emerging thread handling requirements. For this paradigm – hardware multithreading – the manufacturer coined the term SIMT (Single Instruction, Multiple Thread) as extension to the old taxonomy. The SM is capable of tracking many threads without overhead due to context switching. To support balance shifting between vertex and fragment processing loads as described, SMs are designed to handle threads of all different types (vertex, fragment, compute) at the same time.

Warp Processing

A thread group that is launched on an SP is divided into, managed, and executed in arrays of 32 threads. One such array is called *warp* or CTA (Concurrent Thread Array). SMs handle up to 24 such groups concurrently. Upon arrival of work at the SPA, warps are formed of threads of the same program by the SMC and delegated to the SMs. They remain as such until all the threads they contain have finished execution.

The atomic unit of execution is one instruction for a whole warp. Each thread is mapped to an SP. The eight SPs carry out one instruction per cycle, thus take four cycles to complete one instruction for a warp of 32 threads. The scheduling of warps is managed by the SM. Each thread's state, that is register content and program counter are tracked separately. That implicates that not all threads in a group always share the same progress in their program. They might have branched to different paths if the code contains data dependent conditions. The SM can only process threads of a warp in parallel if they share the same execution path. On the other hand, the execution of a warp containing threads with differing program counters must be serialized since all eight SPs can only be issued the same instruction at a time. These threads on different paths are called *branch divergent* [40]. Full efficiency is therefore only given for coherent warps, and if only one thread separates, processing time is doubled.

Therefore, not all threads are always ready for execution and some must be omitted from execution. Besides postponement due to branch divergence, a thread (or a whole warp) may also be blocked waiting for a memory operation to finish. Different warps are executed serially regardless of the state of their threads. Based on warp readiness and instruction type, the scheduler tries to overlap arithmetic operations and memory transactions to maximize throughput. The application can assist this behavior with a high arithmetic intensity and keeping SMs well loaded with threads. Arithmetic intensity here means the overall proportion of arithmetic operations compared to memory transactions in a program [40]. Also incorporated in the scheduler is a fairness mechanism so that no thread is postponed for too long.

Other Multiprocessor Facilities

Besides the SPs and the scheduling logic, SPs feature other facilities. The two SFUs are shared among all eight SPs. They carry out computation of transcendental functions and attribute interpolation (see pipeline description above) and therefore require more than one cycle. SFU operation is separated from the SPs in order not to block them. The scheduler considers this latency when arranging warps [33]. The texture unit serves as a third more specialized processor under the SMC in the SM. It processes groups of four threads of all types at a time. Like the SFUs, execution is out-of-order with the other units. The main reason for this unit is to facilitate texture lookups. A typical texture instruction takes texture coordinates and generates texture addresses with four address generators, then returns filtered samples. Due to the high locality of texture fetches, the unit enjoys a private texture cache. Note that for the 2:1 ratio of SMs to texture units in the TPC, there is no other reason than the consideration of load behavior in typical applications [33].

3.3 Device Memory System

The memory system is divided into multiple areas [40, 44]:

- Beginning on the lowest level of the hierarchy, each SM holds a large set (16384 or 32768) of 32 bit registers for immediate usage during warp execution. They are occupied by threads as their private zero-latency memory. Although shared by the threads running on a SM, communication through registers is not possible.
- The SM uses local memory as a thread-private area to store temporary variables, register content in case of register shortage, and call stack frames. As can be seen in the block diagrams 4, there is no physical representation of local memory on the SM level. It is mapped to the device level DRAM area and is not cached.
- Besides registers, another SM-resident area is shared memory. It provides a low-latency (4 to 6 cycles) way for communication among the threads of an MP. It can also be used to reduce register pressure without the need to fall back to spilling to local memory at high costs. To facilitate handling inter-thread communication, light-weight barrier synchronization instructions with SM-wide effect exist.
- Global memory is in device DRAM and serves all kinds of purposes. All access to DRAM implies high latencies of 400 to 600 cycles. Global memory is uncached, so latencies are high. The host application typically allocates buffers for general purpose compute kernels in global memory. Threads on an SM can use it for infrequent communication if shared memory is reserved for more time critical use. Thread groups running on different SMs should not try to share data via global memory, as the sequence of their execution is not defined.
- Constant memory behaves as global memory except that it can only be written from the host and read from the device where it is cached on SM level for low-latency access.
- Texture memory for GPGPU applications has the same access restrictions as constant memory. It is optimized for locally near access patterns and cached on SM and global level.

Table 3.1 summarizes the different types of device memories together with their properties. Registers, local and shared memory content is lost after a kernel finishes.

Memory	Location	Cache	Access	Scope
Register	SM	-	r/w	Thread
Local	DRAM	No	r/w	Thread
Shared	SM	No	r/w	SM
Global	DRAM	Yes	r/w	All
Constant	DRAM	Yes	r	All
Texture	DRAM	Yes	r	All

Table 3.1: Device memory properties.

Memory access rules

Due to the relatively long access latencies to global memory, most applications would first load operand data to shared memory, compute the results and write them back to global memory. Therefore, global memory access is critical for performance. It can be carried out simultaneously for a whole block of up to 128 bytes if rules for *coalesced* access are met[1] [42]: If all addresses accessed by the threads in a half-warp (that is 16 consecutive threads) lie in a 128B-aligned segment, then a single memory transaction will be issued. If the addresses are spread over a number of such segments, then the according number of memory transactions is required, which results in performance decrease. This mechanism enforces a certain degree of memory locality for problems that should be solved by a group of threads.

Once loaded to shared memory, operands can be accessed about 100 times faster as compared to when global memory resident. Shared memory is laid out into 16 equally sized partitions called *banks* [42]. Addresses that lie in mutually distinct banks can be accessed simultaneously. Banks are arranged so that 16 successive 32-bit words all are located in different banks. Accesses to memory locations in the same bank are called *bank conflicts* and result in serialized transactions.

3.4 The CUDA Programming Model

After a detailed discussion of the underlying hardware, an overview of how it is exposed through the programming model is given.

Overview

To better be able to leverage GPGPU operation, programmers needed an interface abstracting graphics related concepts and bypassing the graphics drivers. Languages like Cg, HLSL, or GLSL are used to write pure shader programs. They are all similar to C but were designed with a focus on graphics pipeline concepts and therefore are largely unattractive to common programmers. Yet before the arrival of commercially available solutions, research projects with the aim to hide graphics peculiarities for GPGPU existed [58]. With the Brook and Sh projects, the streaming programming model was first adapted to GPU hardware with general purpose in mind.

---

[1] The given rule applies to devices of compute capability 1.2 and higher.

NVIDIA CUDA is a C extensions and provides essential abstractions that ease development:

- kernel concept in SPMD fashion
- hiding of massive parallelism
- device memory management

Kernels

Kernels are just like entry points for the host application and must be called with a special syntax. Besides kernels themselves, other device functions can be defined. They can be called just like ordinary C functions. The necessary stack is managed by the SM in local memory. Calls from within kernels of device functions to other functions are allowed, but because of limited device resources, cannot be recursive.

The thread concept already appeared in Section 3.2 in its hardware context. There, threads are grouped into warps for SIMD execution. On the software level, a different thread hierarchy exists and the warp mechanism is abstracted, but its familiarity is recommended as will be explained below.

Threads of the same kernel are combined into blocks and blocks in turn are part of a grid (see upper half of Figure 3.3). Thus, each single thread is assigned a block in a grid and can be identified by the index of its block in the grid and its own thread index in the block. Prior to launching a kernel, input and result buffers must be declared in global device memory, and input data transferred to the device. Upon kernel invocation, the dimensions of the grid and its blocks must be specified, giving the total number of threads to be dispatched. Blocks are then inseparably mapped to SMs where they run until finished. The order of block execution is thereby undefined. Figure 3.3 depicts how a grid of blocks is processed on devices of different processor count. The independence among blocks allows transparent scaling as more SMs accept several blocks which then are handled out-of-order. When assigned to an SM, blocks are split up into warps internally, as discussed earlier.

In the function scope of a kernel, the information about the overall grid and block structure and the thread's location in it is exposed by intrinsic variables. In Listing 3.1, a simple example of a matrix addition is given [40]. Each thread calculates one cell in the result cell. It accesses the operands from the two matrices to be added, based on its own index and its block's index in the overall grid. Note the syntax for kernel invocation taking the grid and block dimensions, called the execution configuration. Blocks can be laid out in height, width and depth, whereas grids are limited to two dimensions.

The arrangement of threads into blocks is an important way of problem subdivision. Blocks should be created with the exploitation of shared memory in mind. Shared memory can only be used for communication between threads of the same block, regardless whether running on the same SM or not.

A simple example for the use of shared memory is *reduction* (see Figure 3.4), the parallel accumulation of an array of values. Single thread complexity is $O(n)$, whereas with parallel computation it can be improved to $O(logn)$. For an array of size $N = 2n$ with adequate n, a thread block of size $N/2$ is set up and computes the result in log $N - 1$ steps. In the first

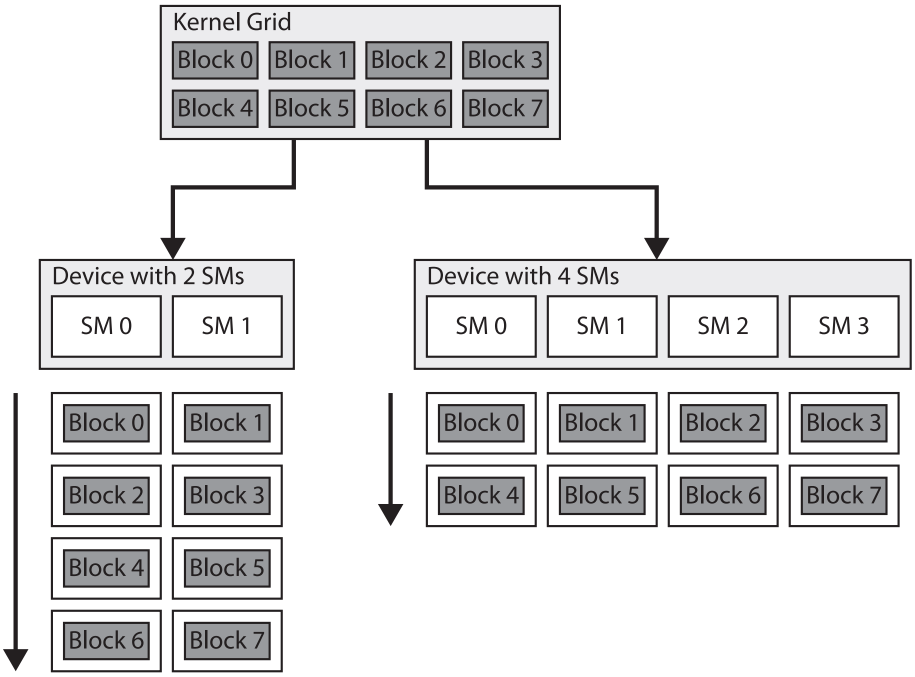

Figure 3.3: Block execution, as in [40].

step, each thread accumulates disjoint pairs of values and saves the result to a shared array of size $N/2$ (one element for each thread), at the position indicated by its own thread index. In the following steps, the threads read and accumulate intermediate results created by other threads from the shared array and update it at their own position. In the end, the first thread sets the overall result [17]. At each step, the number of threads required to do work, is halved, leaving the others idle.

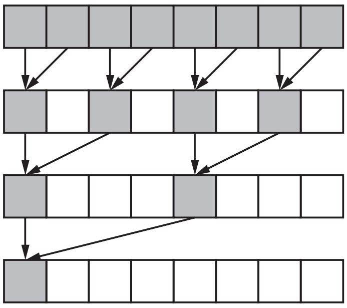

Figure 3.4: Parallel reduction, as depicted in [10].

Note that in this example, problem subdivision with the use of blocks is not illustrated. A comprehensive example of how to subdivide a matrix-matrix multiplication under the meaningful utilization of blocks is given in [40]. To ensure data consistency in the shared array among

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 dimBlock(16, 16);
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x, (N + dimBlock.y - 1) /
        dimBlock.y);
    MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

Listing 3.1: Use of indices in kernel definition.

threads and between steps, barrier synchronization must be applied after each step. It is not ensured that all threads progress evenly in the kernel program. That way, a fast thread one step ahead might read the shared array position of a slower thread, resulting in a read-before-write situation. With synchronization however, each thread is blocked at the barrier instruction until all other threads in the block hit it too.

Performance

Not only must a grid of blocks match the problem, it should also be chosen in a way so that as little as possible device resources lie idle. In the current programming model, only one general purpose compute kernel can run on the device at a time. The occupancy metric is defined as the ratio of actually active warps to the number of possible active warps on the device.

Occupancy is dependent on several parameters (values for low end cards in brackets): the number of available registers per SM (8 kB), shared memory available per SM (16 kB), maximum number of blocks per SM (8), maximum number of threads per block (512), maximum number of active threads per SM (768).

When a grid is distributed to the available SMs, the amount of SM resources claimed by each thread (registers, shared memory) together with block size determine how many blocks are assigned per SM. Occupancy is related to performance through the following effects:

- Multiple blocks per SM hide thread inactivity caused by barrier synchronization. While one block is engaged in synchronization and ends up with a small number of active threads, another block might be ready to keep the hardware busy.

- A high number of threads per block give many warps the scheduler may choose from while others are inactivated waiting for memory transactions or other costly instructions.

After all, choosing the execution parameters for a kernel means balancing occupancy for latency hiding and overall resource utilization [42].

There are more examples of how knowledge of device details can be integrated for better performance.

Warp-level synchronization is an effect that makes barrier synchronization superfluous among threads of the same warp. Warps are atoms of execution on the SM and its threads will never be rearranged, provided that they follow the same program branch. This can be exploited in some situations [42].

3.5 Discussion

CUDA does not expose a steep learning curve but demands considerable tuning and knowledge of the underlying hardware if near optimal performance and good device occupancy are expected. Several extensions to and modifications of the model, both hardware and software, have been proposed.

One advantage over other programming models is that explicit data packing before vector instruction execution is not required, i.e. operands need not to be lined up in contiguous memory locations. Although, ordered access increases performance as described in the previous section, applications not too dependent on memory transactions still benefit from the ability to access any memory location, a great gain in simplicity [9]. Difficulties may arise from the way how data structures must be mapped to the provided grid and block schema. This is easy for matrices and volume cubes (fluid dynamics). There are however applications that, even if exposing a high degree of data-parallelism, are difficult to adapt to (e.g. graph traversal algorithms).

Data access patterns must be organized according to the user's algorithm to benefit from the available memory hierarchy, especially from shared memory. A number of enhancements could promote the use of shared memory, which is currently restricted in some ways. No producer-consumer relationships among thread blocks are possible because of the transient nature of shared memory. Such a relationship can only exist between kernels that relay data in global memory. These patterns would be facilitated if shared memory content were to be preserved between kernel calls. Also, the ability to spawn kernels from already running kernels has been proposed for this purpose [9].

Currently, the parallel computing landscape is quickly evolving. On the GPU sector numerous different architectures are evolving side by side. The common challenge for the next years is to increase the bandwidth between CPU and GPU. Several signs suggest a unification of CPU and GPU in the long run. Although canceled in late 2009, Intel's Larrabee architecture, for example, intended to combine strong features of today's multi-core CPUs and current many-core GPUs, i.e. the coherent cache hierarchy, x86 instruction compatibility and wide SIMD properties. Also, AMD already began the single die integration of GPU and CPU with devices from its Fusion series CPUs for mobile applications [44, 55]. NVIDIA's release of the successor architecture to the GT200 series[2], called Fermi, is the next step in scale. Structurally equal, it brought a number of enhancements in hardware, such as scaled up features like warp width and number of SPs per SM, EMI capability, and a revised version of the CUDA programming model.

---

[2] Used for this thesis.

CHAPTER 4

Related Work

This chapter tries to give an overview of existing research that can be related to the subject of this thesis. References treating GPU technology and the matter of speech coding have already been given in the according sections. Although no publications about GPU speech coding exist, many treat associated subproblems. Some discuss the potential of general purpose GPUs, others present novel implementations of speech coding algorithms, also under the aspect of parallelization.

While in recent times, GPU potential was often praised, there are also critical voices indicating its overestimation. In [31], GPU and CPU are compared regarding their architectures. It is shown that if CPU programs were written with the same attention regarding optimization according to their architecture as GPU code usually is, then the performance differences between the two may turn out much lower than many publications claim it to be.

Signal processing via GPGPU is covered well. Most of the GPGPU applications for signal processing are in the context of computer vision and video encoding due to inherent data parallelism. Also not audio related, [54] demonstrates the usefulness of GPGPU in DSP using the example of a 2D convolution operation. Further, advantages of commodity market developments are pointed out.

Audio-related GPGPU applications on the other hand are rare. In [53] for example, the audio signal is not processed directly, but the GPU is rather employed for audio rendering based on room acoustics models. As part of the discussion of further applications in audio, the authors identify the GPU's inefficiencies when confronted with short tap filters (as occur in speech encoding), but praise performance gains of 60,000 taps parallel FIR implementations. The parallel encoding of multiple speech channels, which is the intention of this thesis, is mentioned as a potentially feasible application of GPGPU in the conclusion of the article.

[12] presents a real-time GPU implementation of the convolution operation for large input vectors. Programming is via a graphics API.

Several publications exist that deal with the optimization of speech encoding algorithms similar to that used in the TETRA environment. All of these efforts however are targeted towards

DSP platforms. The optimization of a DSP implementation of the G.729 (CS-ACELP) codec is treated in [6].

The only article directly related to the TETRA speech codec is [14]. It too describes an optimized DSP implementation. It advertises the benefits of hand-coded assembler in contrast to relying on compiler support.

One early article directly addressing the mapping of CELP algorithms on parallel platforms (old vector machines) is [50]. Included is a data flow diagram and it is pointed out which portions of the algorithm demand most of the processing. Unfortunately, actual hints on how parallelism in CELP could be exploited are not given.

A more recent work treating CELP-style algorithms and parallel processing is [25]. It presents a chip design that integrates a vector accelerator unit into a scalar RISC architecture and measures the performance by an implementation of the G.729 CS-ACELP codec. In its introduction it brings forward as a motivation that there is a demand in VoIP networks for gateways that encode and consolidate many speech channels. The authors come to the conclusion that their approach of parallel acceleration with an SIMD width of 16 is useful. Parallelism is drawn from loops with fixed-point arithmetic without cross-iteration dependencies (filters, ...). However, details about where exactly in the encoding procedure the vector operations cause an increase in performance, is not revealed. Another attempt to accelerate G.729 is given in [36]. It aims at hardware/software partitioning of the procedure and presents results from a MATLAB simulation. Acceleration is mainly by exploitation of task parallelism, i.e. the simultaneous processing of independent stages.

Besides publications explicitly treating parallel aspects of speech codecs, there are also those dealing with related subproblems.

In [37], FIR and IIR filter operations are implemented with the use of the AltiVec low SIMD-width extension for general purpose CPUs. It is shown that IIR operations provide almost no parallelism and thus can hardly be accelerated. Further, it is pointed out that theoretical performance gain due to SIMD width is hard to reach because of the necessary data reorganization overhead, and the upper bound for common DSP algorithms is estimated to only 1.6 to 11.66. A similar approach is [28] on using the Intel SSE SIMD extension. [26] presents a method that allows the decomposition of IIR filters into multiple lower order IIR filters. Carried out in the context of a VLSI design study, the thesis on which [49] is based treats the problems of common speech coding operations regarding parallelism. The inefficiency with parallel implementations of recursive filters is pointed out and the parallel calculation of products in multiply-accumulate patterns is identified as the only way for acceleration.

While the above studies on filters are hardly applicable to the GPU, [56] introduces a well suited parallelization approach for FIR filter operations and served as the basis for some considerations in this work.

ACELP codebook searches resemble tree searches and are responsible for a considerable part of the algorithm's complexity. While [2] and [21] treat tree searches in general, [29] introduces the method as recommended for the TETRA coder by ETSI. Parallelization of tree searches is treated in [57] utilizing SIMD, in [47] on streaming platforms, and in [19] in the context of multi-processor environments.

CHAPTER 5

Implementation

The aim of the implementation efforts is to determine how many channels can be processed by a modern GPU (results for GeForce 8400 GS). In Chapter 2, the basis for an understanding of the encoding algorithm was provided. The adaption to the target platform now requires the analysis of its procedural structure, the actual data flow, and a detailed breakdown into individual building blocks. In this work all these considerations as well as the implementation itself are based on the reference implementation associated with the ETSI specification document [22]. The reference code is written in ANSI-C and highly structured[1]. All signals involved in the algorithm are represented in various 16-bit fixed point formats. The code served as a starting point for all own implementation experiments and greatly facilitated output verification.

5.1 Encoding Procedure

The following diagram (Figure 5.1) shows the structure of the sequential reference implementation and shows all functional blocks together with signal routing. Most of the blocks correspond to single functions in the ETSI C-code.

First of all, the raw speech signal is preprocessed. That includes offset compensation (DC portion removal) and division by 2 to avoid saturation during synthesis filtering.

(1) Next is the LPC analysis for the whole frame. As the basis for the LPC equation system, the ACF must be calculated. For spectral smoothing, the resulting autocorrelation signal is multiplied with a lag window. The LPC coefficients are obtained from the Levinson-Durbin algorithm that solves the system.

(2) For an efficient transmission of LPC coefficients, they are transformed to LSP representation which offers easier quantization and interpolation. During frame processing both quantized and unquantized sets of coefficients are needed. In LSP interpolation, the current LSP values and those which were determined in the previous frame are used to calculate a set of unquantized and

---

[1] There are functions for each arithmetic operation, obviously to facilitate adaption to DSP hardware. This makes the code very hard to read and understand.

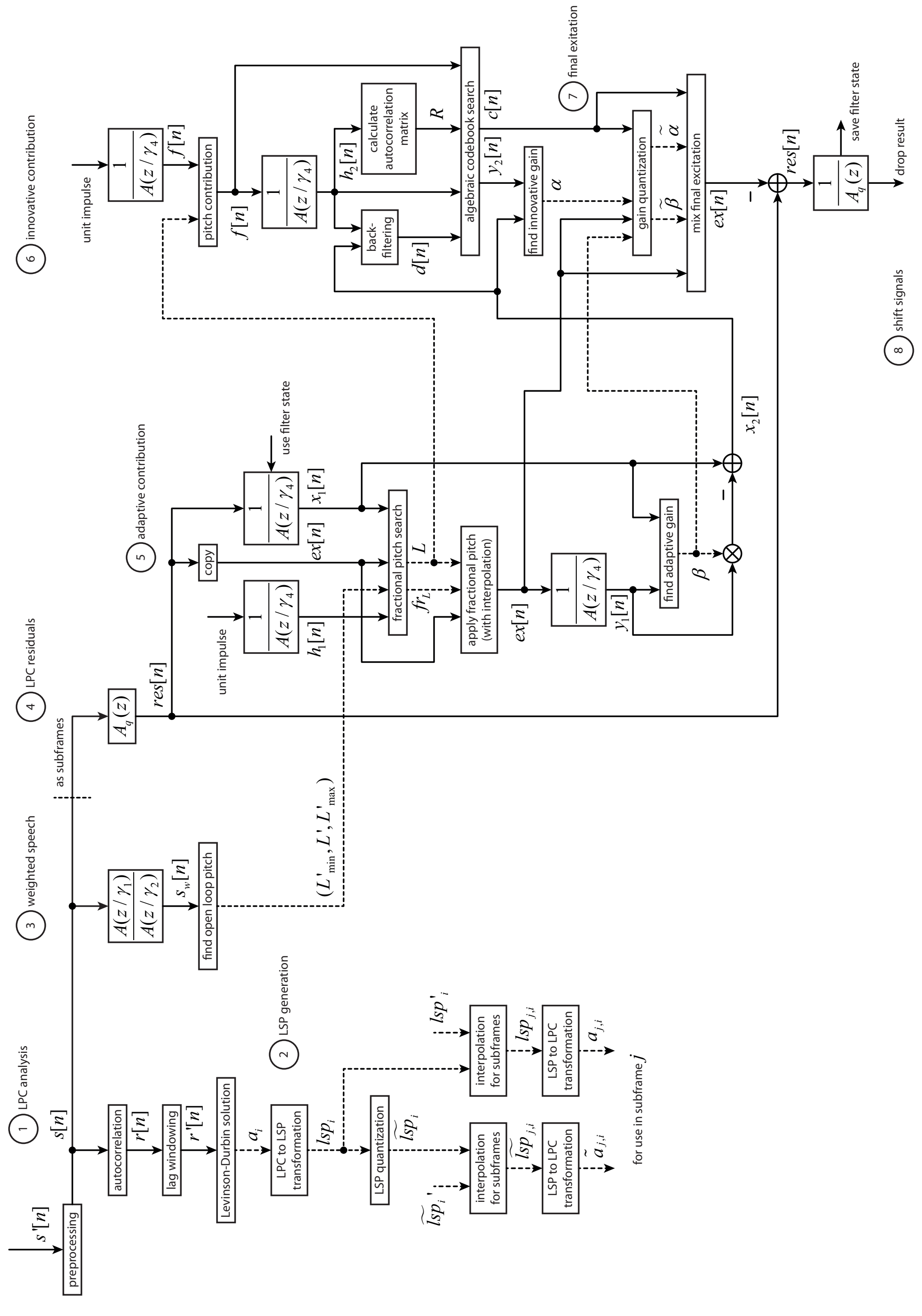

Figure 5.1: Complete data flow diagram of the TETRA encoder, according to reference implementation.

quantized LSP for the use in each subframe. The quantized LSP are now ready for transmission, and the current LSP values must be saved for the interpolation during the next frame. Since the decoder relies entirely on the LSP representation which it receives, the encoder must, too, base all the filtering on LSP to preserve consistency. This is part of the AbS principle. Therefore, it is necessary to once again retransform all the LSP to LPC in order to obtain the (time domain) coefficients for frame processing.

(3) In the last step of whole-frame processing, a weighted speech signal is formed by successively applying the analysis and synthesis filters to the speech signal using spectrally expanded coefficients. This is to be done in four (number of subframes) successive steps because the four sets of unquantized LPCs that resulted from LSP retransformation must be used as available. Thereby, each of the synthesis filter passes relies on the filter state of the previous step. The resulting signal is the basis of the preliminary open loop pitch analysis stage. The lag estimation and its range bounds are used in the closed loop pitch search of the first subframe.

(4) Thereafter, subframe processing commences. According to the CELP procedure, this involves the search for the adaptive and the innovative contribution that make up the excitation signal for the decoder. First, the LPC residual signal is calculated by analysis filtering with quantized coefficients. The result is the ideal excitation signal and therefore must serve as the basis for target signals in subsequent stages.

(5) The essential step in finding the adaptive contribution is the fractional pitch search. It calculates the normalized correlation and searches it for a maximum in the vicinity (L'_{min} and L'_{max}) around the lag estimation L' provided by the open loop pitch search. Other required signals for this step are: the impulse response $h_1[n]$ of the synthesis filter, the excitation signal $ex[n]$, which is a copy of the LPC residuals, and the target speech signal $x_1[n]$ to compare against (the synthesis filtered LPC residuals). In each subframe the resulting lag parameters replace those estimated in the previous subframe (or those from the open loop pitch search, in the case of the first subframe).

In the next step the resulting parameters, the integer lag L and its chosen fraction $fr_L \in \left\{-\frac{1}{3}, 0, +\frac{1}{3}\right\}$, are applied to the excitation signal $ex[n]$ which now corresponds to the unit gain vector from the adaptive codebook. It is yet to be scaled to approximate the target speech signal $x_1[n]$. Therefore, once again synthesis is simulated to produce the GSVQ shape vector $y_1[n]$. Now, the required pitch gain β that scales $y_1[n]$ to best match $x_1[n]$ can be calculated. The last step in this part is to actually apply the scale factor β to $y_1[n]$ which produces the optimal speech resembling signal (note: anonymous in the diagram) that can be generated by means of the adaptive codebook alone. If subtracted from the target $x_1[n]$ a differential speech signal, $x_2[n]$, is obtained. It expresses what the adaptive codebook still failed to contribute and what the innovative contribution is meant to compensate.

(6) As a basis to the algebraic codebook search, the pitch affected impulse response $f[n]$ is calculated as stated in Section 2.9. This is then synthesis filtered producing the signal $h_2[n]$ from which the autocorrelation matrix R is derived. Additionally, a version of the target signal $x_2[n]$ backward filtered with $h_2[n]$ is needed for the search. The results of the search are the speech signal $y_2[n]$ and the excitation signal $c[n]$, the selected vector from the algebraic codebook. The difference signal $x_2[n]$ resulting from the adaptive contribution is then again used as target vector in the calculation of the gain α that scales $y_2[n]$. The results from this stage are the code vector

$c[n]$ and the gain α.

(7) In the end, the results of both contributions $ex[n]$ with β and $c[n]$ with α are used for gain quantization. The final excitation signal is mixed from $ex[n]$ and $c[n]$ with quantized gains $\tilde{\alpha}$ and $\tilde{\beta}$. In order to base the filtering in the next frame on the proper filter state, the difference signal $res[n] - ex[n]$ is once more filtered and the resulting states saved. With the saved filter state, filtering in the next subframe can be continued as if there were no frame partitions.

5.2 Leveraging GPU Properties

To get a feeling of how the GPU copes with the load imposed by the above procedure, the reference code was modified so that it can be run on the GPU as a single thread without making use of parallel processing. Thereby, the whole encoder was launched as a single CUDA kernel to keep host/device memory transactions low. On the test system, an NVIDIA GeForce 8400 GS[2] was compared to an Intel Core2 Duo E8400[3] CPU. The results were fairly disappointing and proved the platform's poor adequacy concerning sequential processing. A three-second speech sample took the GPU 5.1 seconds. For comparison, it took a standard desktop CPU 0.14 s for the same task. This is a factor of roughly 36 against the GPU. Above all, the GPU's performance isn't even real-time compliant. Arguably, these results encourage questioning the eventual purpose of the whole endeavor.

However, it is normal that a given reference code must be modified to match whatever platform it should be adapted to. [13] and [6] are examples of similar efforts. Both give hints on optimizing similar reference C code for speech codecs like the G.729 ACELP, targeting DSP platforms. As a first step the reasons for the poor measured performance should be identified.

As stated in Chapter 3, the GPU is developed for processing video data which, unlike audio data, inherently exhibits a high degree of parallelism. That means that for GPUs sequential throughput (fast processing of small data sets) is not the primary issue, and extra efforts are required to achieve it. The GPU unfolds its strength when confronted with large data sets that may be processed at moderate rates.

The actual technical reasons for the poor sequential performance are:

- Without any further measures, all signals reside in global device memory. As stated in Chapter 3, global memory latency is high (400 to 800 cycles [42]), and complex calculations with frequent access to certain buffers therefore stall completely.

- Unlike common CPUs, the GPU does not feature automatically managed caches. It is the responsibility of the programmer to properly use shared memory in the same role. For efficient loading of shared memory, a number of threads are needed to ensure coalesced access, as explained in Chapter 3.

- Even if assuming a well managed shared memory allocation, complicated access patterns to shared memory still cause bank conflicts and the serialization of access operations. In complex calculations as in this speech coder, the effect cannot always be avoided.

---

[2] 567 MHz core, 1400 MHz shader, 400 MHz memory

[3] 3.0 GHz

- Poor instruction mapping. For example, multiply-accumulate instructions exist on the GPU but could not be found in compiled assembler output files.

The first step is to leverage the properties of the GPU to enable real-time operation and reach lowest possible latencies. Only then one can consider evaluating the number of channels the GPU can manage in parallel. This requires analysis of how parallelism can be created from the given procedure.

5.3 Exploiting Parallelism

As described in Chapter 3, warps in a thread group may be postponed because waiting for a memory transaction or the result from an arithmetic operation. Since the execution of instructions for warps is serialized, this leads to latencies. The internal thread scheduling unit takes care of a balanced instruction flow and will select waiting warps with ready instructions for execution while others are postponed. Thus, the loss caused by inevitable latencies can be reduced by providing enough instructions (preferably data independent). This mechanism is called *latency hiding* and is the key for an efficient parallel implementation. Exploiting the power of the GPU means providing enough work for the threads on the multiprocessors.

The measure for how effectively the multiprocessor can be kept busy is called *occupancy* [40]. It is defined as the ratio of the number of active warps to the maximum number of active warps (24 warps for devices with compute capability 1.1, 32 warps for devices with compute capability 1.3).

For an implementation of the coder, the aim is to find an assignment of work to CUDA threads that combines high occupancy with real-time compliance. A single speech channel produces data at 480 bytes each 30 ms. Real-time requirements demand that processing must be finished in less than 30 ms.

With such little input data, reasonable occupancy values cannot be reached. Above all, the next speech frame does not even exist at the time the current frame is processed. Moreover, as has been shown in the description of the encoder procedure, processing of frames is a highly sequential task. Calculation of LPC coefficients, codebook contributions, and subframes (interlinked through filter states) are all interdependent in that each produces parameters for successive stages.

Data and progress independence however is given among different speech channels. All this suggests drawing the required occupancy from the simultaneous processing of more than one channel on a single MP.

Since use of shared memory must be maximized and communication over slow global memory be minimized, it is certain that a channel is assigned to a single MP.

Therefore, occupancy can be gained in two ways:

1. Many threads cooperate during frame processing for one channel.

2. Multiple channels are processed on a single MP simultaneously, thus interleaving instructions from both channels.

Both ways can be combined. That means that a certain number of threads cooperates in the processing of one channel, still leaving MP resources because of low occupancy. To better utilize the MP, another thread group processes a second channel, on the same MP.

The work loads which thread groups are assigned to are the distinct stages from the encoding algorithm (e.g. LPC solution, synthesis filtering, codebook searches), sometimes called functional units in the following. As will be seen, the degree of possible thread cooperation (thread group size) varies from unit to unit.

While mapping more than one channel onto an MP does certainly extend the overall latency, it does so to a far less degree compared to the serial processing of frames, as depicted in Figure 5.2. For a runtime of T for a single channel, the overall runtime for n channels on one MP is expected to be far less than nT due to latency hiding (without overloading the device). That way, for a device with N multiprocessors, the real-time processing of Nn channels can be ensured. The highest number of multiprocessors in current single-die devices (NVIDIA's GT200 family) is 30.

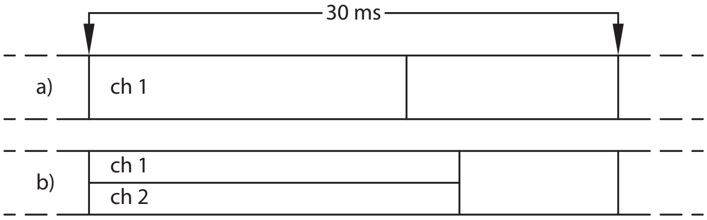

Figure 5.2: Single (a) and multiple (b) channel processing per MP within the frame time.

Note that in this work, only data parallelism is considered. In the CUDA version used here, just a single thread can be launched on the GPU. The device is then blocked until all threads finished execution [40]. For the processing of functional units, that means that all thread groups must process the same stage for their channel, and different groups cannot be assigned diverse work loads. Therefore, task parallelism cannot be exploited and is not treated.

The introduction of parallelism in the coder is possible on different levels (from fine to coarse):

1. Parallel processing of single functional units. Sufficiently data parallel operations can be executed by a number of threads, e.g. the amplification of a signal by a certain factor with each thread processing a single sample.

2. Exploitation of data parallel structures within the encoding procedure. Parallel processing of multiple functional units of the same type without dependencies, in the form of different CUDA blocks on the same MP. Example: multiple independent filter passes in parallel.

3. Depending on the outcome of levels 1 and 2, the number of channels that can be processed per multiprocessor can be determined. It is possible that a single MP has enough resources (required registers per thread, shared memory) to process multiple channels at the same time, i.e. assign its threads to different input streams and still maintain reasonable latencies (through latency hiding).

4. The result of level 3 leads to the final number of channels that can be handled on device-level. This is application-inherent data parallelism.

Before the above points can be answered, it must be determined which functional blocks in the algorithm are worth the effort for parallel implementation or optimization. Obviously, those blocks that take the most runtime are considered good candidates for analysis.

5.4 Unit Ranking

In order to identify the most runtime intensive units, a complexity ranking was arranged. Therefore, a single-thread GPU implementation with each functional unit as a separate CUDA kernel was implemented and profiled with a 3s speech sample (101 frames). Figure 5.3 shows the first ten functional blocks ordered according to their runtime proportion for all 101 frames.

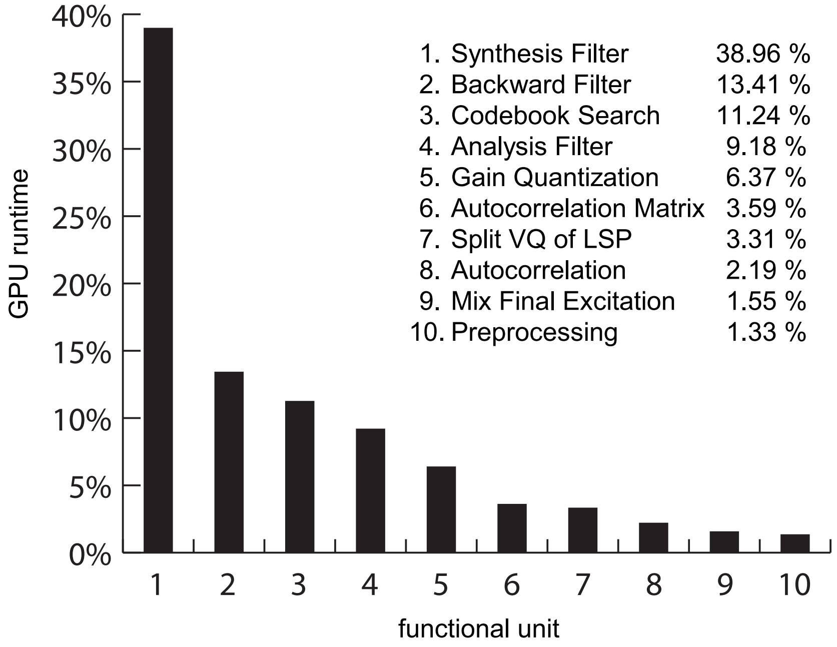

Figure 5.3: The ten most processing intensive functions and their average runtime proportion per frame.

One note on the results of the ranking. The runtime distribution among units is relatively even. That means that several units require modifications to achieve a significant performance gain. The ten units included in the ranking account for over 90 percent of the overall frame processing time.

5.5 Selected Units

The aim of the parallel implementation is to reduce the overall runtime of frame encoding as much as possible. Since single-thread performance on the GPU is poor (see above), ways of efficient parallel processing must be found for each stage.

Thereby, the method is mainly that of exploiting independent iterations in loops, i.e. parallel processing of unrolled loops. In cases where this is impossible, due to dependent iterations, significant gain may be achieved by extraction and parallel preparation of intermediate results from calculations that otherwise must be left sequential.

For example, a minimum among the results of distance functions has to be searched. The original loop would iterate through a set of elements, calculate the distance function, and update the minimum value if the current distance is smaller than the previously found minimum. In parallel processing, the intermediate results, the distances, can be calculated in advance, stored (which requires some amount of additional memory) and searched for the minimum afterwards. Dependent on the specific case, this may reduce overall runtime greatly. Depending on the number of distances, the search for the minimum can be done sequentially (for small sets), or by parallel reduction (for large sets), as has been introduced in Chapter 3.

This section treats in detail those units that were implemented for the streaming platform. Although all of the ranked units were analyzed for parallelization, only a handful were implemented: Analysis filter, synthesis filter, backward filter, algebraic codebook search, gain quantization, the calculation of the autocorrelation matrix which is employed in the codebook search, and the split VQ of LSPs after conversion from LPC coefficients.

All units share the same way of loading input data into shared memory. Coalesced access to global memory always requires a certain number of threads. [ref from nvidia]

Things to consider in each unit implementation:

- kernel handling in overall arrangement:
- memory access patterns
- shared memory employment, loading
- warp divergence, synchronization

Three among the four most expensive units, namely synthesis-, analysis- and backward-filter (that make up for about 60% of the runtime) are all of similar form and should be discussed first.

Analysis Filter

The analysis filter $A(z)$ is of all-zero form[4] and therefore can be implemented as FIR filter

$$y\left[n\right] = \sum_{i=1}^{10} a_i x\left[n-i\right]$$

---

[4] The transfer function's denominator polynomial is 1, so there are no poles.

with $n = 0, ..., 59$ (subframe length) where x must be accessible 10 samples into its past (so 70 samples for x) . Since there are only ten coefficients, the number of iterations needed for one output sample is already low. The work is therefore best assigned to 60 threads, with thread n calculating the result in $y[n]$. For the tests, $x[n]$, a_i as well as $y[n]$ were stored in shared memory. According to shared memory recommendations in [40] the simultaneous access to a_i causes bank conflicts resulting in the serialization of instructions and some performance loss. Further optimizations addressing memory access were omitted since the results from parallelization were already satisfactory:

Function name:	Residu	
Runtime of unmodified kernel (1 thread):	$446\mu s$	(9.18% runtime)
Runtime parallel (60 threads):	$11\mu s$	

Synthesis Filter

The synthesis filter $\frac{1}{A(z)}$ is of all-pole form[5] and therefore must be implemented as IIR filter. Its calculation is sequential by nature because each iteration depends on the results of the previous:

$$y\left[n\right] = \sum_{i=1}^{10} a_i y\left[n-i\right]$$

The GPU does not cope well with this form. Measurements have shown that one filter pass takes about $200\mu s$ which is the main cause for high latency in frame processing (see ranking). If this iterative form is retained, parallelism can only be exploited on a very small scale: each iteration, a vector consisting of the 10 previous results is multiplied with the vector of coefficients. This vector multiplication can be carried out simultaneously by ten threads. Since reduction doesn't pay off with 10 elements, a single thread must then be used to sum up the products. This preparation of products in the loop reduces the filter runtime to $72\mu s$ with only 10 involved threads. The findings on whether and how filter operations can be further accelerated on the given platform are discussed in the following.

NOTE: [ACustomVLSIArchitectureFor] treats this issue

Recursive filters and parallel processing

Only a few articles dealing with the parallelization of recursive filters can be found. In [26] for example, the decomposition of IIR structures into multiple IIR structures of lower order is introduced. The aim is to promote parallel processing in low-latency DSP applications. However, the described transformation is not suitable in the context of this work since it introduces errors and seems to be rather suited for IIR filters of much higher order than 10.

An exact approach targeted at short SIMD platforms such as Intel SSE is presented in [28]. To eliminate data dependencies that prevent parallel processing, algebraic transformations of the algorithm, as proposed in [46], are applied. The described technique with fused filter taps is suited for filters with an order of multiples of the platform's SIMD width (four in the case of

---

[5] The nominator polynomial is 1 so there are no roots.

Intel's SSE). For the speech coder with a filter order of 10 and an atomic vector width of 32 on the GPU (CUDA warp size) little gain can be expected.

Apart from these approaches, the synthesis filter can be transformed on a much higher level. According to common knowledge, an IIR filter with truncated impulse response can be exactly represented by a non-recursive FIR form by interpreting the IIR impulse response $h[n]$ as the transfer function of the equivalent FIR filter. The filter operation is then the convolution of the input signal with $h[n]$:

$$y\,[n] = \sum_{i=0}^{n} x\,[i]\,h\,[n-i]$$

with $n = 0, ..., 59$. Since in its IIR form not more than 60 elements (subframe width) are calculated, this means that the impulse response is effectively truncated to length 60, justifying the substitution with the FIR form. In the case of the speech coder this condition is met. Both input signal and transfer function have 60 elements.

Scalar convolution as defined here is not well suited for parallel processing. On the GPU, if threads are assigned as seen in the implementation of $A(z)$, work distribution is unbalanced: the first thread calculates only a single product, but the last thread sums up 60 products. That means that compared with the untruncated convolution of $A(z)$, the critical path remains unchanged. For an efficient parallelization an even distribution of work must be found.

In [56], a general decomposition algorithm for discrete computational problems is proposed. For demonstration in the article, an efficient parallel convolution method is introduced. What makes this method interesting is that it succeeds in subdividing the convolution into a number of smaller operations of equal complexity. The novelty therein seems to have lain in the disjunction of subproblems, which makes the method well suited for parallel processing.

The procedure comprises three steps: decomposition, solution of subproblems and reconstruction. Based on integer modular arithmetic, the involved signals $x[n]$ and $h[n]$ (here both of dimension $N = 60$) are decomposed into their residue classes under a chosen modulo $p \geq 1$ which is an integer factor of N. Let, for example, $p = 5$ so that $x[n]$ and $h[n]$ decompose into $N/p = 12$ sequences $x_i[k]$ and $h_j[k]$ of length p given by

$$x_i[k] = x[kp+i]$$

$$h_j[k] = h[kp+j]$$

with $0 \leq i, j \leq 11$ and $0 \leq k \leq 4$ resulting from the chosen p.

Thereafter, $(N/p)^2 = 144$ subproblems $y_{i,j}$ for all combinations of input sequences $x_i[k]$ and $h_j[k]$ can be specified as convolutions of much shorter length p, in the form

$$y_{i,j}[k] = \sum_{l=0}^{p} x_j\,[l]\,h_i\,[k-l]$$

The calculation of these intermediate results $y_{i,j}[k]$ requires much less iterations and, equally important, can be solved independently, in parallel.

Reconstruction is by summing up elements from $y_{i,j}$ which yields the exact result of the original convolution operation

$$y[n] = \sum y_{i,j}[k]$$

where the index constraint

$$i + j + k\frac{N}{p} = n$$

applies with $k \in (0, ..., p-1)$. This constraint is not given explicitly in [56] but is consistent, more succinct, and involves n directly. To better illustrate the implications of the given constraint, the table in Figure 5.4 shows how the intermediate results $y_{i,j}$ contribute to the overall $y[n]$. For $0 \leq n < N/p$ there are $n+1$ summands whereas for $N/p \leq n < N$ there are N/p elements that contribute to $y[n]$.

n	0	1	2	...	5	...	12	13	...	29	...
$y_{0,0}$	$y_{0,0}[0]$						$y_{0,0}[1]$				
$y_{0,1}$		$y_{0,1}[0]$						$y_{0,1}[1]$			
$y_{1,0}$		$y_{1,0}[0]$						$y_{1,0}[1]$			
$y_{1,1}$			$y_{1,1}[0]$								
$y_{0,2}$			$y_{0,2}[0]$								
$y_{2,0}$			$y_{2,0}[0]$								
...				...		...			...		...
$y_{0,0}$					$y_{2,3}[0]$					$y_{2,3}[2]$	
$y_{0,1}$					$y_{3,2}[0]$					$y_{3,2}[2]$	
$y_{1,0}$					$y_{1,4}[0]$					$y_{1,4}[2]$	
$y_{1,1}$					$y_{4,1}[0]$					$y_{4,1}[2]$	
$y_{0,2}$					$y_{0,5}[0]$					$y_{0,5}[2]$	
$y_{2,0}$					$y_{5,0}[0]$					$y_{5,0}[2]$	
...				...		...			...		...

Figure 5.4: Some contributions of $y_{i,j}[k]$ for $p = 5$ and $N = 60$. Each column n holds elements that accumulate to $y[n]$. Columns for $n \geq 12$ are displayed incomplete to preserve the expressiveness of the table. Layout inspired by [56].

Thread assignment in the GPU implementation is obvious: $(N/p)^2$ threads share the calculation of $y_{i,j}$ as given above. Thereby, an explicit decomposition phase to build x_i and h_j is omitted and the required operands are accessed directly from local memory resident input sequences, according to the rule. No efforts were undertaken to avoid shared memory bank conflicts. Yet, care has been taken to ensure a proper storage layout for $y_{i,j}$, enabling uncomplicated access during the reconstruction phase. Storage is laid out as suggested in the table in Figure 5.4 so that elements to accumulate are located with a constant stride of N apart from each other. For the accumulation, N threads are assigned so that each thread accumulates to one element $y[n]$ of the final result. The accumulation can be left unbalanced since at most N/p summands occur. With this thread assignment, there are two distinct phases employing a different number of threads: calculation of $y_{i,j}$, and reconstruction to $y[n]$. While suboptimal, no efficient way has been found to employ all N/p^2 threads during reconstruction.

Considering parallel execution, theoretical computational complexity decreases by p^2 for the first part [56] and is of $O(N)$ for reconstruction. Experiments have shown that $p = 5$ works best for the used target platform and the subframe length of 60. Lower p means more subconvolutions, leading to longer runtime for reconstruction. Higher p means longer operands for subconvolutions.

Function name:	Syn_Filt	
Runtime of unmodified kernel (1 thread):	$480\mu s$	(38.96% runtime)
Runtime with product extraction (10 threads):	$72\mu s$	
Parallel decomposition algorithm (144 threads):	$15\mu s$	

Application in the coder and implications

Needless to say, the required IIR by FIR substitution cannot be applied without any modifications to the coder. In the original design (see diagram in Figure 5.1), the required impulse response $h[n]$ representing $\frac{1}{A(z)}$ for the current subframe must be found anyway (signal $h_1[n]$ in the diagram). The necessary parameters for the calculation of $h_1[n]$ are already known before subframe processing commences: the unit pulse and the spectrally expanded, unquantized, and interpolated LPC coefficients, $a_{j,i}$ in the diagram. $h_1[n]$ can therefore be calculated in advance by IIR filtering of four passes in parallel (4) for each subframe. The same holds for the (augmented) impulse response $f[n]$ needed for the determination of the innovative contribution. This can be done by decomposed FIR filtering since impulse responses are already available. Each subframe, $\frac{1}{A(z)}$ occurs seven times. The only place in subframe processing where decomposed FIR filtering cannot be applied is synthesis filtering of the LPC residuals. This is because at this point, the impulse response is first determined under the use of the previously left filter memory. For an estimation of costs for the required filter passes see Chapter 6.

Backward Filtering

Backward filtering is convolution of an input signal with a reversed transfer function h, again, with both signals of the same length. Instead of explicitly reversing h before filtering, the access to the signal buffer is adapted to yield the desired result:

$$y\,[n] = \sum_{i=n}^{60-1} x\,[i]\,h[i-n]$$

This operation is well suited for the application of the decomposed convolution algorithm given before. No additional preparations are required since the impulse response is available anyway. The algorithm must merely be modified to ensure inverted access to h for the backward filtering effect.

Function name:	Back_Fil	
Runtime of unmodified kernel (1 thread):	$1309\mu s$	(13.41% runtime)
Parallel decomposition algorithm (144 threads):	$24\mu s$	

Algebraic Codebook Search

The codebook search is the most complex and critical stage in the coder because it minimizes the mean squared error between the input speech and the synthesized speech. A sparse code vector from the set A_k has to be found that when further shaped by the lag-afflicted impulse response f provides an adequate contributive excitation signal. To tell the adequacy of a given vector, the autocorrelation matrix R and the backward filtered target vector $d[n]$ are used: the optimal vector A_k is the one that maximizes the term

$$\frac{\rho_k^2}{\varepsilon_k}$$

which is the quotient of the squared crosscorrelation between code- and target vector

$$\rho = \sqrt{2}d\left[m_0\right] - d\left[m_1\right] + d\left[m_2\right] - d[m_3]$$

and the cumulative energy

$$\varepsilon = 2R\left[m_0, m_0\right]$$

$$+R\left[m_1, m_1\right] - 2\sqrt{2}R\left[m_0, m_1\right]$$

$$+R\left[m_2, m_2\right] + 2\sqrt{2}R\left[m_0, m_2\right] - 2R\left[m_1, m_2\right]$$

$$+R\left[m_3, m_3\right] + 2\sqrt{2}R\left[m_0, m_3\right] + R\left[m_1, m_3\right] + 2R\left[m_2, m_3\right]$$

with m_i being the particular pulse position of pulse i from the range given in the table. The code vector is not explicitly given but appears as indices into $d[n]$ and R.

The algebraic code vectors consist of 2 to 4 pulses of alternating sign with the first pulse amplitude of $\sqrt{2}$, the others of unit amplitude. Pulse positions are as given in Table 5.1 below

Codebook parameters	Pulse Positions	bit allocation
Amplitude: $+1.4142$	$0, 2, ..., 56, 58$	5
Amplitude: -1	$2, 10, 18, 26, 34, 42, 50, 58$	3
Amplitude: $+1$	$4, 12, 20, 28, 36, 44, 52, (60)$	3
Amplitude: -1	$6, 14, 22, 30, 38, 46, 54, (62)$	3
global sign flag		1
shift flag		1

Table 5.1: ACELP pulse positions of the code vectors used in the TETRA codec, as in [22].

In the sequential implementation as advocated in [22], the search is conducted in four nested loops with each loop traversing all positions of one specific pulse. This is a depth-first search of a tree with $30 \cdot 8 \cdot 8 \cdot 8 = 15360$ nodes corresponding to all combinations of possible pulse positions. For performance reasons, the search is kept non-exhaustive. The method that causes unpromising pulse combinations to be skipped is called *focused search* [29]. Two threshold values that must be exceeded by $abs(\rho)$ exist. Pulse positions m_2 and m_3 will only be tested if the preceding two pulses cause high enough $abs(\rho)$. Otherwise the next combination of m_0 and m_1 is tested. This leads to variable search times per subframe. To control the maximum number of nodes searched, an additional counter is decremented for each search involving pulses 2 or 3. The focused search procedure is presented in the listing below.

```
counter = 350
for all 30 possible positions of pulse 0:
   add pulse 0 contribution to correlation and energy
   for all 8 possible positions of pulse 1:
      add pulse 1 contribution to correlation and energy
      if correlation > threshold #1:
         for all 8 possible positions of pulse 2:
            counter := counter - 3
            if counter <= 0 finish
            add pulse 2 contribution to correlation and energy
            if correlation <= threshold #2:
               for all 8 possible positions of pulse 3:
                  counter := Counter - 4
                  if counter <= 0: end search
                  add pulse 3 contribution to correlation and energy
                  save pulse position if quotient is new maximum
```

Listing 5.1: Focused codebook search.

In the absence of appropriate information, it can be assumed that on average each threshold is passed in 50% of cases, i.e. $((((30 \cdot 8)/2) \cdot 8)/2) \cdot 8 = 3840$ leaf nodes present adequate pulse combinations, of which only the first $350/((4+3)/2) = 100$ are actually inspected. Since the codebook search accounts for a fair proportion of runtime in the coder, ways for parallel processing are of particular interest. The problem of parallel tree searches has been extensively treated, mainly in the context of game-playing AI [2], [21]. While it was found that SIMD architectures can successfully be employed for tree searches [57], [47] is the only article directly dealing with tree searches on streaming platforms. Although much smaller in scale as most of the problems treated in the literature, the parallelization of the algebraic codebook search involves the same related issues as discussed in [19]. It deals with the principal implications of parallel search, however in the context of multi-processor environments.

A naive parallel solution to the search was implemented. It partially neglects the tree structure and views each code vector as an equal element, without considering shared pulse positions among combinations. The procedure is as follows:

1. 240 threads are assigned the 240 combinations of pulse 0 (30 positions) and pulse 1 (8 positions). Each thread calculates in parallel the parameters ρ and ε according to its assigned pulse positions.

2. If the correlation from the combination passes the first threshold, then the according thread approaches the remaining subtree, i.e. additional 64 combinations of pulse 2 (8 positions) and 3 (8 positions). Otherwise the thread terminates. Each continuing thread sequentially searches the remaining pulse combinations on the basis of its own combination of pulse 0 and 1, and keeps track of the local maximum as occurs in the subtree[6]. For the 240 remaining subtrees (of which just a portion is searched), the maxima are stored in shared memory together with essential parameters identifying the resulting vector (the path of the node in the tree).
3. Now, the highest of the 240 maxima is searched by a single thread. The parameters (i.e. pulse positions) stored with the same index as the maximum allow the reconstruction of the actual codebook vector.

The parallelization of this unit bears some potential, however, the search space seems to be relatively small in comparison to the SIMD width of the GPU.

Function name:	D4i60_16	
Runtime of unmodified kernel (1 thread):	$3540\mu s$	(11.24% runtime)
Parallel naive search (240 threads):	$270\mu s$	

Gain Quantization

After the establishment of the adaptive and innovative contributions, the associated gains β and α are quantized as a tuple using predictive VQ [20]. The procedure is as follows [22]: The actual energy of each contribution is predicted as a linear combination of energies from past subframes. To reach a better quantization performance, not the energies are quantized directly but the errors resulting from their predictions (they have smaller variance). Quantization is by means of a 6-bit codebook with codeword length 2 for the tuple (α, β). The actual quantized energies are then simply defined as the predictions with quantized errors added. The quantized gains as they result in the decoder are a function of quantized gains and contribution energies, [22].

The processing intensive parts here are the calculation of energies and – to a lesser degree – the codebook search. The procedure requires the energies of the impulse response of $1/A(z)$, which can be found by one of the parallel solutions given above. Furthermore, impulse responses of the adaptive contribution and the innovative contribution are needed. The energy of a signal $x[i]$ is given by

$$\sum_{i=0}^{59} x[i]^2.$$

60 elements for an SIMD width of 32 are too little to justify accumulation by reduction. Again, the only efficient measure in this situation is to calculate the squares $x[i]^2$ in advance and leave the accumulation sequential.

Additionally, energies of adaptive and innovative contributions do not depend on each other and can be calculated in parallel.

---

[6] Subtree here means set of pulse combinations.

The final codebook search of only 64 tuples is ill-suited for parallel processing. Similar to the above problem of aggregation, the best here is to calculate distances in parallel and then search the minimum sequentially.

The overall parallelization gain of this unit is low.

Function name:	Ener_Qua	
Runtime of unmodified kernel (1 thread):	$1116\mu s$	(6.37% runtime)
Runtime (60 threads):	$207\mu s$	
Runtime with parallel distances(60 threads):	$180\mu s$	

Autocorrelation Matrix

The algebraic codebook search makes use of the autocorrelation matrix derived from the impulse response $f[n]$ of A(z/γ) with applied lag in the form $f[n] + f[n - L]$. Due to the layout of possible pulses on even positions only, it is sufficient to calculate the matrix on 32 positions, resulting in a symmetric 32x32 matrix [22]. Each cell in the matrix represents the sum of the autocorrelation for one even and its successive odd element in f:

$$r_{i,j} = \sum_{k=0}^{31-i} \left(f\,[2j]\, f[2(i+j+k)] + f\,[2j+1]\, f[2\,(i+j+k)+1] \right).$$

A naive approach to a parallel implementation is to assign each thread with the calculation of one of the matrix' 32 diagonals. That means summing up the products along the diagonal and store each temporary sum accordingly. This corresponds to the progress of index j in the above equation. This assignment however is inefficient due to its load imbalance: thread 0 must calculate 32 cells, while thread 31 calculates only a single cell. Summation for one cell is inherently sequential and cannot efficiently be parallelized, especially not with vector sizes smaller or equal the SIMD width.

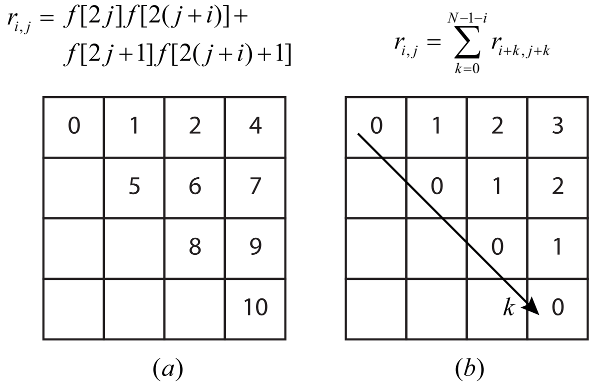
$r_{i,j} = f[2j]f[2(j+i)]+$
$f[2j+1]f[2(j+i)+1]$
$r_{i,j} = \sum_{k=0}^{N-1-i} r_{i+k,j+k}$
0
1
2
4
5
6
7
8
9
10
0
1
2
3
0
1
2
0
1
k
0
(a)
(b)

Figure 5.5: Illustrative matrix of size 4, numbers indicating thread assignment. (a) Preparation of products, balanced. (b) Stepwise summation of products in diagonals, unbalanced.

The only means, as in the enhanced version of the recursive IIR filter (see Section 5.5), is to prepare the products occurring in the autocorrelation. Each thread would then calculate only one cell in the matrix without other dependencies than to elements in $f[i]$, and therefore can be balanced (diagram (a) in Figure 5.5). Summation of diagonals must be left unbalanced because the sum in each cell is dependent on its predecessor in the diagonal. Diagram (b) in Figure 5.5 illustrates the imbalanced division of work among the threads. On the small scale of only 32, no gain can be expected that would justify a balancing mechanism. Moreover, 32 elements are too little for the application of parallel reduction.

A possible cell distribution for a number of threads that equals the matrix dimension in multiple passes is: For $i \leq j$, cells $r_{i,j}$ are processed in pass p:

- threads $t = p, ..., N - 1$ calculate cell $r_{(t,p)}$
- threads $t = 0, ..., p - 1$ calculate cell $r_{(N-p+t,N-p)}$

Function name:	Cal_Rr2	
Runtime unmodified kernel (1 thread):	$1170\mu s$	(3.59% runtime)
Runtime unbalanced parallel (60 threads):	$47\mu s$	
Runtime with product preparation (60 threads):	unimplemented	

Split VQ of LSP

Section 2.9 already introduced LPC to LSP conversion and explained its significance in the encoding process. The LSPs are not quantized directly. To ease implementation, quantization is carried out in cosine domain, that means that ω_i are transformed to $q_i = cos(2\omega_i/8000)$ [22].

The bit assignment for split VQ is as follows:

Codebook 1 (256 vectors): q_1, q_2, q_3

Codebook 2 (512 vectors): q_4, q_5, q_6

Codebook 3 (512 vectors): q_7, q_8, q_9, q_{10}

As in previous situations, exploitation of data parallelism means to calculate the vector distances in parallel, and in a following stage conduct the search for the index to the minimum distance. For a search of 512 elements and SIMD width of 32, employing parallel reduction will hardly pay off. Also, after the parallel calculation of distances, a threefold parallel search for minima won't speed up the procedure significantly, and in addition to that would require more shared memory space.

If more than one channel should be processed per multiprocessor, not all 512 threads can be assigned. As can be seen from the results, increasing the number of threads yields unsatisfactory gains. The potential for parallel processing in this unit is mediocre.

Function name:	Clsp_334	
Runtime of unmodified kernel (1 thread):	$1322\mu s$	(3.31% runtime)
Runtime parallel (128 threads):	$313\mu s$ (no reduction)	
Runtime parallel (256 threads):	$303\mu s$ (no reduction)	

CHAPTER 6

Overall Results

This section summarizes the results from optimization efforts for the the modules given in Section 5.5. As discussed in Section 5.3, simultaneous processing of more than one channel per MP is promising. All implemented optimizations however are single-channel variants. Due to the effort involved, the multi-channel variants and an integration to an executable encoder package had to be skipped in this thesis. Moreover, the use of the decomposition algorithm for recursive filter substitution would require substantial modifications to the coder and the test environment.

As described in Section 5.5, several functional units have been studied for parallelization. For evaluation purposes, the most promising units have been implemented and performance was measured for the encoding of a single speech channel on a single MP. Thereby, the number of cooperating threads varies from unit to unit, but does not exceed the maximum number of active threads per MP. It is even low enough allowing the possible simultaneous processing of more than one channel per MP.

To tell how the single-channel optimizations influence the GPU performance for the encoder, a projection of the overall runtime is provided.

Table 6.1 lists the results. The first column holds the runtime proportions from the single-thread GPU test run of the complete encoding procedure, including those stages which were not reviewed for optimization. The second column contains factors for speedup through parallelization. This is the runtime of the single-thread measurement divided by that of the optimized variant[1] (in μs). This applies to all items except the synthesis filter (which is therefore italic in the table).

For the synthesis filter, the structural modification required for the decomposition algorithm is now assumed (see the description in Section 5.5). Two different implementations are needed: The recursive one (slow) to find the impulse response for a given set of coefficients and filter state, and the convolution variant (fast) that uses the impulse response for calculation. The final speedup factor of 20 is composed in the following manner. The synthesis filter is invoked 32 times per frame. Thereof, 4 times on frame level and 28 times on sub-frame level, that is 7

---

[1] Note that here the results are not at all compared to CPU performance.

Unit	Runtime [%]	Speedup	Threads
Analysis filter	9.18	40	60
Synthesis filter	38.96	20	144
Backward filter	13.41	55.5	144
Codebook search	11.24	13.1	240
Gain quantization	6.37	6.25	60
Autocorrelation matrix	3.59	25	60
LSP VQ	3.31	4.34	128
others	13.94	1	1
Total	100	5.22	max. 240

Table 6.1: Module runtime proportions before optimizations and according gains.

passes per sub-frame. On frame level, 1 IIR pass is required to acquire the impulse response and 4 FIR passes for the actual operation. On sub-frame level, 1 IIR and 6 FIR passes are required, all in all 4 IIR and 24 FIR passes. The projected speedup is then the average of 5 IIR and 28 FIR passes (33 passes), which is 20 according to the results in Section 5.5 with $72\mu s$ per IIR and $15\mu s$ per FIR pass, in contrast to the $480\mu s$ from the single-thread variant.

The third column in the table indicates the number of required threads as given in the according implementation results. It is required to tell the maximum number of employed threads per channel, and allows the determination of occupancy.

Besides the reviewed and implemented units, the item "others" subsumes all unreviewed encoder parts for which no speedup is accounted for. These account for about additional 18 units of varying complexity. At the bottom of the table, the sum of runtime percentages, the weighted average of speedup factors and the maximum number of deployed threads are given. Figure 6.1 illustrates the shift in runtime distribution among the functional units before and after optimizations.

As measured by the runtime tests with the three-second sample that took 5.1 seconds, the projected runtime for the same input sequence is therefore $\frac{5.1}{5.22} = 0.98$. This corresponds to about $\frac{1}{3}$ of frame time for one channel. Accordingly, a single MP is capable of encoding three channels in real-time considering the sequential approach with poor occupancy.

Thus, an optimization enabling real-time compliance succeeded, and a projection for a total number of channels can be given.

At the time of writing, [43] holds the latest list of available NVIDIA devices. The largest single-die device holds 30 multiprocessors, so with the above projection, 90 channels could be encoded in real time. Devices with more multiprocessors exist but cannot be attributed to the class of mainstream commodity products, and are therefore ignored for now. Test runs of implemented functional units with a GT200 series card (GTX275) revealed that in comparison to the low-end GeForce 8400 GS, there is no significant difference in performance of the multiprocessor units. Cards indeed differ regarding the number of multiprocessors on the die and – according to reviews – graphics processing features. However, compute kernel operations in multiprocessors take the same time. Both cards share the same clock frequency for the shaders.

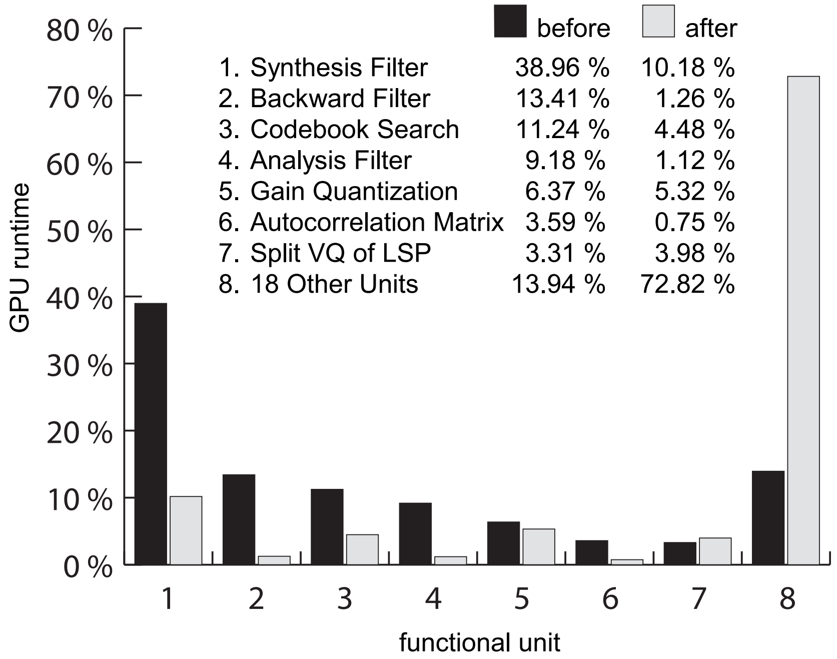

Figure 6.1: The most processing intensive functions and their average runtime proportion per frame, together with projected runtime after optimizations.

Unit	8400 GS	GTX 275
Multiprocessors	2	30
Cores on device	16	240
Active warps (per MP)	24	32
Registers (per MP)	8k	16k
Shared memory (per MP)	16kB	16kB
Core Clock (MHz)	567	633
Shader Clock (MHz)	1400	1404
Memory Clock (MHz)	400	1134
Device Memory (MB)	512	896
Memory Interface (bit)	448	448

Table 6.2: Comparison of cards used in tests, [38, 39, 43].

This suggests that performance in the speech coding task is dependent on the shader clock only, which gives evidence on the compute-bound nature of the problem. See Table 6.2 for details on the cards used in tests.

A further step is still the maximization of occupancy on the device. With the given optimization, no more than 240 threads are required per channel, as outlined in Table 6.1. 240 threads correspond to 8 warps of 32 threads each. The maximum number of active warps is 32 for the device class with 30 multiprocessors. That means that with at most 8 warps assigned to a channel, four channels must be processed simultaneously to reach an occupancy of 100%. If the processing of four channels with increased good occupancy still fits into the $30ms$ time frame, then the total channel capacity of the same device is extended to 120 channels.

An Intel Xeon quad-core CPU with hyper threading is also capable of encoding 120 channels in real-time, as tests have shown. Although more efficient, the CPU could be combined with a GPU in the accelerator role. The advantage is that such a system could be deployed as a single unit without the use of specialized hardware.

CHAPTER 7

Conclusion

This thesis ventures a first step towards the application of massively parallel computing platforms for speech coding purposes. Experiments are based on the TETRA speech codec, which is an example of the representative CELP method. It is shown that a CELP coder is a compute-intensive procedure which inherently exhibits only limited data parallelism and a small problem size in contrast to other GPGPU applications. Modern GPUs concentrate high computing power, however unfold their strength only if provided with enough work. Single-thread capabilities on the GPU have shown to be poor for several reasons such as high memory latencies and manually managed cache. Since the encoding procedure must be optimized in several places to reach real-time operation, some load can be gained from parallelization of the most complex encoder stages. Thereby the focus lies mainly on filtering operations, for which satisfying parallel variants were introduced. The majority of code had to be left in single-thread operation, and could further be accelerated with little effort. In the speech processing context, more of the required volume for the GPU can be provided when considering the encoding of whole batches of channels simultaneously. This is because the GPU holds a number of mutually independent multi-processors to which the processing of channels is assigned. With the device class used in this work, it was found that by employing the most powerful commodity product (manufacturer NVIDIA), the GPU is capable of encoding at least 90 channels in real time. Compared to high-end desktop CPUs, which are capable of encoding about 120 channels, this is disappointing. However, in this stage of development, the GPU could be used as an accelerator relieving the CPU from a good portion of its work. The benefit from this would be the possibility to deploy a single compact unit capable of acting as a complete gateway in the TETRA network. Even multi-GPU systems are available (e.g. NVIDIA Tesla) for high performance computing. For gateways that are required to encode a high number of channels, they could be used as self-contained units and still save processor load for other tasks.

Chances are high that the number of channels the GPU is able to encode can be increased by further optimizing the available load per multiprocessor, thus exploiting the mechanism called *latency hiding* to a higher degree. This would be achieved by the simultaneous processing of input data from different channels on a single MP, opposed to the sequential approach as has been implemented so far.

Yet, many aspects of the subject remain unexplored. The reason for this is partly the tedious development process[1] that makes quick experiments with new ideas almost impossible. All modifications have to be thoroughly tested, which requires constant expansions to the test environment.

At the time of writing the device family this work is based on is already succeeded by even more powerful architectures. Due to lack of time, none of these devices could be tested for encoding performance.

Currently the latest architectures show a trend towards bigger but fewer multiprocessors. The list of device properties in the appendix of the latest CUDA programming guide [43] shows this development in GPU architecture regarding SIMD-width and number of incorporated multiprocessors. Less MPs house more and more compute cores. While previous MPs had 8 cores, those in recent designs contain as much as 48. As a result, the atomic logic vector width of 32 for previous devices has been extended to 64. From what has been learned during the encoder parallelization, this already exceeds the SIMD vector width adequate for low bit-rate coders such as the TETRA. The reason for this design decision by the manufacturer seems clear: why should precious die area be wasted to redundant infrastructure (schedulers, memory connections on MP-level) if problem-inherent data parallelism (for graphics and large computational tasks) is high enough so that SIMD vector width can be further increased for better utilization.

On the other hand, new multiprocessor designs seem to focus on more versatile scheduling and branch handling, which could turn out as an advantage for speech coding [41].

Next Steps

For the further development of a completly GPU-based TETRA encoder, a number of issues should be considered. First, in order to be able to tell the real effect of latency hiding in multichannel mode, the runtime-intensive units must be implemented for the processing of not just one channel at a time, but n. The result thereof will be the knowledge of how each algorithm stage profits from the effect. Second, the operational prototype should be integrated into an actual TETRA infrastructure for field tests, and as a proof of concept.

---

[1] See the reference code from [22].

Bibliography

[1] Tor Aamodt. Architecting graphics processors for non-graphics compute acceleration. In *2009 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Special Session on Computer Architecture*, Computer Graphics Proceedings, Annual Conference Series, pages 963–968. IEEE, IEEE, 2009.

[2] Selim G. Akl, David T. Barnard, and Ralph J. Doran. Design, analysis, and implementation of a parallel tree search algorithm. In *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, pages 192 – 203, March 1982.

[3] B. Atal and J. Remde. A new model of lpc excitation for producing natural-sounding speech at low bit rates. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '82.*, volume 7, pages 614 – 617, may 1982.

[4] B. S. Atal and S. L. Hanauer. Speech Analysis and Synthesis by Linear Prediction of the Speech Wave. *J. Acoust. Soc. Am.*, 50(2):637–655, August 1971.

[5] Bishnu S. Atal and J. R. Remde. A new model of lpc excitation for producing natural-sounding speech at low bit rates. *Proceedings of the 1982 International Conference on Acoustics, Speech, and Signal Processing*, 1(12):614–617, 5 1982.

[6] M.T. Awan, S. Masud, N. Khan, and F. Abdullah. Multilevel optimization of speech coding algorithms for modern dsp architectures. In *Communications, Computers and signal Processing, 2005. PACRim. 2005 IEEE Pacific Rim Conference on*, pages 265 – 268, aug. 2005.

[7] H.Y. Su C. Laflamme, J.P. Adoul and S. Monssette. On reducing computational complexity of codebook search in celp coder through the use of algebraic codes. In *Proc. of IEEE Int. Conf. on Acouctics, Speech, and Signal Processing*, pages 937–940, April 1990.

[8] Jr. Campbell, Joseph P. and T. E. Tremain. Voiced/unvoiced classification of speech with applications to the u.s. government lpc-10e algorithm. *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 67(12):473–6, 1986.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[10] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Proceedings of the 2008 Symposium on Application Specific Processors*, pages 101–107, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Wai C. Chu. *Speech Coding Algorithms: Foundation and Evolution of Standardized Coders*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[12] Brent Cowan and Bill Kapralos. Spatial sound for video games and virtual environments utilizing real-time gpu-based convolution. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, Future Play '08, pages 166–172, New York, NY, USA, 2008. ACM.

[13] David H . Crawford and Emmanuel Roy. Implementing speech coding algorithms. *Embedded Systems Programming Magazine*, pages 39 – 52, February 2002.

[14] Heetal Kiran Devendra. Real time implementation of tetra speech codec on tms320c54x, 2002.

[15] Jack Dongarra, Dennis Gannon, Geoffrey Fox, and Ken Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, 3(1), 02/2007 2007.

[16] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972.

[17] Jens-Fabian Goetzmann. Massively parallel contact simulation on graphics hardware using nvidia cuda. In *Informatiktage*, pages 251–254, 2008.

[18] N. Görtz. *Aufwandsarme Qualitätsverbesserungen bei der gestörten Übertragung codierter Sprachsignale*. Shaker Verlag, Aachen, 1999.

[19] A. Grama and V. Kumar. *A Survey of Parallel Search Algorithms for Discrete Optimization*, chapter 11. Addison Wesley, 2003.

[20] D. Haoui, A.; Messerschmitt. Predictive vector quantization. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '84*, pages 420 – 423, Mar. 1984.

[21] Holger Hopp and Peter Sanders. Parallel game tree search on simd machines. In *Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems*, IRREGULAR '95, pages 349–361, London, UK, 1995. Springer-Verlag.

[22] European Telecommunications Standards Institute. ETSI EN 300 395-2 v1.3.1 - Terrestrial Trunked Radio (TETRA); Speech Codec for full-rate traffic channel; Part 2: TETRA codec, January 2005.

[23] P. Kabal and R. P. Ramachandran. The computation of line spectral frequencies using Chebyshev polynomials. *IEEE Transactions on Acoustics Speech and Signal Processing*, 34(6):1419–1426, 1986.

[24] A. M. Kondoz. *Digital Speech: Coding for Low Bit Rate Communication Systems*. John Wiley & Sons, New York, NY, USA, 2 edition, 2004.

[25] K. Koutsomyti, S. R. Parr, V. A. Chouliaras, and J. Nunez. Applying data-parallel and scalar optimizations for the efficient implementation of the g.729a and g.723.1 speech coding standards, 2005.

[26] A. Krukowski, I. Kale, , and G.D. Cain. Decomposition of iir transfer functions into parallel arbitrary-order iir subfilters. In *Proceedings of the IEEE Nordic Signal Processing Symposium (NORSIG '96)*, pages 175–178. IEEE, IEEE, September 1996.

[27] Jakub Kurzak, Alfredo Buttari, Piotr Luszczek, and Jack Dongarra. The playstation 3 for high-performance scientific computing. *Computing in Science and Engineering*, 10(3):84–87, 2008.

[28] R. Kutil. Parallelization of iir filters using simd extensions. In *Systems, Signals and Image Processing, 2008. IWSSIP 2008. 15th International Conference on*, pages 65 – 68, June 2008.

[29] C. Laflamme, J.-P. Adoul, R. Salami, S. Morissette, and P. Mabilleau. 16 kbps wideband speech coding technique based on algebraic celp. *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 0:13–16, 1991.

[30] Daren Lee, Ivo Dinov, Bin Dong, Boris Gutman, Igor Yanovsky, and Arthur W. Toga. CUDA optimization strategies for compute- and memory-bound neuroimaging algorithms. *Computer Methods and Programs in Biomedicine*, December 2010.

[31] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38:451–460, June 2010.

[32] J.S. Lim and A.V. Oppenheim. Enhancement and bandwidth compression of noisy speech. *Proceedings of the IEEE*, 67(12):1586–604, 1979.

[33] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[34] J. Makhoul. Linear prediction: A tutorial review. *Proceedings of the IEEE*, 63(4):561–580, 1975.

[35] Satoshi Matsuoka, Takayuki Aoki, Toshio Endo, Akira Nukada, Toshihiro Kato, and Atushi Hasegawa. Gpu accelerated computing - from hype to mainsstream, the rebirth of vector computing. In *Journal of Physics: Conference Series 180 (2009)*, 2009.

[36] A. Nassery, M.R. Ghajar, S. Najafzadeh, and B. Khajehnejad, M.A. andForouzandeh. Evaluation of hardware/software partitioning of conjugate-structure algebraic celp (cs-acelp) algorithm using matlab. In *Information and Communication Technology, 2007. ICICT '07. International Conference on*, pages 287 – 290, March 2007.

[37] Huy Nguyen and Lizy Kurian John. Exploiting simd parallelism in dsp and multimedia algorithms using the altivec technology. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 11–20, New York, NY, USA, 1999. ACM.

[38] NVIDIA. Geforce 8400 gs specification. `http://www.nvidia.com/object/geforce_8400.html`. Accessed: 16/04/2011.

[39] NVIDIA. Geforce gtx 275 specification. `http://www.nvidia.com/object/product_geforce_gtx_275_us.html`. Accessed: 16/04/2011.

[40] NVIDIA. Cuda programming guide 2.3. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`, 2009.

[41] NVIDIA. Nvidia fermi whitepaper. `http://www.nvidia.com/object/IO_86776.html`, 2009.

[42] NVIDIA. Cuda best practices guide 3.0. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf`, 2010.

[43] NVIDIA. Cuda programming guide 3.2. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`, 2010.

[44] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[45] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 425–432, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[46] J. Robelly, G. Cichon, H. Seidel, and G. Fettweis. Implementation of recursive digital filters into vector simd dsp architectures. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, pages V – 165–8 vol.5, May 2004.

[47] Kamil Rocki and Reiji Suda. Parallel minimax tree searching on gpu. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, pages 449–456, Berlin, Heidelberg, 2010. Springer-Verlag.

[48] M. R. Schroeder and B. A. Atal. Code-excited linear prediction (CELP): High quality speech at very low bit rates. In *Proc. of IEEE Int. Conf. on Acouctics, Speech, and Signal Processing*, pages 937–940, March 1985.

[49] P. D. Schuler, R. H. S. Hardy, and V. Cuperman. A custom vlsi architecture for low-delay speech coding. In *Proceedings of the Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference*, ICASSP '91, pages 1161–1164, Washington, DC, USA, 1991. IEEE Computer Society.

[50] M. SCHULTHEISS and A. LACROIX. Mapping celp algorithms onto parallel architectures. *Proc. Douzieme Colloque sur le Traitement du Signal et ses Applications (GRETSI)*, 1989.

[51] B. Soong, F.; Juang. Line spectrum pair (lsp) and speech data compression. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '84*, pages 37 – 40, Mar. 1984.

[52] D. Tarjan and K. Skadron. Multithreading vs. streaming. In *MSPC '08: Proceedings of the SIGPLAN Workshop on Memory Systems Performance and Correctness, March 2, 2008*. MSPC, 2008.

[53] Nicolas Tsingos. Using programmable graphics hardware for acoustics and audio rendering. In *Audio Engineering Society Convention 127*, 10 2009.

[54] Alexander S. van Amesfoort, Ana Lucia Varbanescu, Henk J. Sips, and Rob V. van Nieuwpoort. Evaluating multi-core platforms for hpc data-intensive kernels. In *Proceedings of the 6th ACM conference on Computing frontiers*, CF '09, pages 207–216, New York, NY, USA, 2009. ACM.

[55] Vasily Volkov and James Demmel. State of the art report on gpu. Technical report, Visualization and Virtual Reality Research Group, School of Computing - University of Leeds, 2009.

[56] S.R. Wang and P. Siy. Parallel-decomposition algorithm for discrete computational problems and its application in developing an efficient discrete convolution algorithm. In *Vision, Image and Signal Processing, IEE Proceedings -*, pages 40 – 46, Feb 1995.

[57] Dan Wu, Pan Chen, Kui Dai, Jinli Rao, and Xuecheng Zou. Implementation of parallel game tree search on a simd system. In *Information Engineering (ICIE), 2010 WASE International Conference on*, pages 62 – 66, Aug. 2010.

[58] Enhua Wu and Youquan Liu. Emerging technology about gpgpu. In *IEEE Asia Pacific Conference on Circuits and Systems, 2008. APCCAS 2008.*, pages 618–622. IEEE, 2008.