Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2011

Nuno Alexandre Sarmento Antunes

Desmodulação para RF Utilizando Compressed Sampling com GPUs

"No problem is too small or too trivial if we can really do something about it."

— Richard P. Feynman

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2011

Nuno Alexandre Sarmento Antunes

Desmodulação para RF Utilizando Compressed Sampling com GPUs

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre de Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica de José Manuel Neto Vieira, Professor do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president	**Prof. Doutor Tomás Oliveira e Silva** Professor Associado da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)
vogais / examiners committee	**Prof. Doutor José Manuel Neto Vieira** Professor Auxiliar da Universidade de Aveiro (orientador)
	Prof. Doutor Rui Pedro de Oliveira Alves Professor Auxiliar da Universidade de Aveiro (co-orientador)
	Prof. Doutor José Manuel Tavares Vieira Cabral Professor Auxiliar da Universidade do Minho

agradecimentos / acknowledgements

Desejo oferecer os meus agradecimentos e saudações às seguintes pessoas, pelo contributo dado durante a criação desta tese: Ao professores José Manuel Neto Vieira e Rui Pedro de Oliveira Alves pela orientação e apoio dado até ao último momento. À Professora Isabel Duarte pelas bases teóricas para as simulações inciais e revisões subsequentes. E finalmente aos meus amigos e familiares que me apoiaram durante a duração e desenvolvimento do projecto e da dissertação.

Resumo

Dadas as necessidades modernas de transmissão de dados, coloca-se a questão de implementar métodos de aquisição de dados que sejam mais flexíveis que as aplicações actuais, e que proporcionem desempenhos iguais ou superiores. Esta dissertação apresenta um desmodulador aleatório que permite realizar a aquisição de sinais amostrados a frequências sub-Nyquist, e um método de recuperação destes sinais utilizando Orthogonal Matching Pursuit(OMP). Uma implementação do algoritmo OMP utilizando a capacidade de processamento paralelo de um Graphical Processing Unit (GPU) é demonstrada e comparada com uma implementação padrão em CPU.

Os resultados mostram que a implementação paralela em GPU apresenta melhorias significativas em termos de velocidade se comparada com a implementação em CPUs. Isto pode tornar a solução apelativa para o objectivo de recuperação de sinais em tempo real.

Abstract

Given the modern requirements for data transmission and aquisition, there is a ever-present research for data aquisition methods that are more flexible than current solutions, and that can match or outperform their performance. This thesis presents a random demodulator that can acquire signals sampled at sub-Nyquist frequencies, and a method to recover the sinal using Orthogonal Matching Pursuit(OMP). An implementation of the OMP algorythm that uses the parallel processing power of a Graphical Processing Unit(GPU) is then demonstrated and benchmarked against a reference CPU implementation.

The results suggest that a parallel implementation using GPUs presents significative improvements against the CPU one regarding the speed of data recovery. This solution shows that it is suitable to recover signals acquired using a random demodulator in real time.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ALU	Arithmetic and Logic Units
BLAS	Basic Linear Algebra Subprogram
CPU	Computer Processing Unit
CS	Compressed Sensing
CUBLAS	Compute Unified BLAS
CUDA	Compute Unified Device Architecture
FLOPS	Floating Point Operations Per Second
GPGPU	General Purpose GPU
GPU	Graphic Processing Unit
MIMD	Multiple Instructions, Multiple Data
OMP	Orthogonal Matching Pursuit
SDR	Software Defined Radio
SIMT	Single Instruction, Multiple Threads
SM	Stream Multiprocessors(SM)
SP	Stream Processors

T&L	Transform and Lighting
TPC	Thread Processing Cluster
TS	Thread Scheduler

Capítulo 1

Introdução

As últimas décadas têm sido caracterizadas por um aumento da quantidade de informação transmitida através de rádio-frequência. De um meio que maioritariamente transportava informação analógica a partir de um número limitado de emissores, encontram-se agora uma grande quantidade de dispositivos capazes de transmitir e receber informação, espalhados através do espectro das frequências disponíveis. Torna-se então necessário procurar novos métodos de transmissão de dados que optimizem a largura da banda disponível, e isso significa procurar frequências de transmissão mais elevadas e métodos de aquisição de sinais mais flexíveis.

Uma das soluções apresentadas encontra-se no campo dos *software-defined radio* ou SDRs, que efectuam aquisição de uma grande quantidade do espectro de rádio frequência e posteriormente o tratam em *software*. No entanto, estes estão limitados pela operação clássica da transformação do sinal analógico em digital e dos respectivos limites impostos à largura de banda. Neste aspecto importa relembrar o teorema da amostragem ou de Shannon-Hartley – para um sinal contínuo $x(t)$ de frequência máxima f_{max} Hertz, a reconstrução do sinal só está garantida se a amostragem for efectuada a uma taxa superior a $W = 2f_{max}$. No entanto, nos sistemas SDR é frequente ser necessário converter para o domíno digital sinais de rádio frequência com uma grande largura de banda. Para tal é necessário utilizar conversores com uma elevada frequência de amostragem em que os

problemas de "jitter" do relógio utilizado condicionam o número efectivo de bits que se consegue obter.

Uma nova aproximação a este problema utiliza Compressed Sensing (CS), um método matemático com raizes na ciência estatística e que pode ser aplicado com sucesso na conversão de sinais analógicos para o domínio digital. Esta técnica consiste em multiplicar o sinal que se deseja capturar por um sinal pseudo aleatório de alta frequência, sendo o sinal resultante integrado e amostrado a uma razão inferior. Se efectuarmos esta operação com vários estádios em paralelo, é possível reconstruir o sinal original a partir de misturas e utilizando algoritmos de optimização. Ao mover esta tecnologia para o campo da amostragem em tempo real de um sinal. Na Secção 2.3 é apresentado um desmodulador aleatório para rádio frequência baseado nesta técnica e inspirado no artigo de Tropp [15]. Esta técnica apresenta a grande vantagem de permitir utilizar conversores analógico digital com uma menor taxa de amostragem e por consequência com um relógio com menos *jitter*.

Embora o processo de amostragem digital possa ser implementado com eficiência, a obtenção dos sinais está matematicamente limitada pela resolução de um problema subdimensionado da forma $y = Ax$. Uma solução para este problema passa pela utilização do algoritmo *greedy* denominado Matching Pursuit, descrito na Secção 2.4. Uma optimização deste algoritmo denominado por Orthogonal Matching Pursuit (OMP), que garante reconstrução do sinal, foi de seguida estudado em detalhe.

Apesar de algoritmos como o OMP serem bastante eficientes na resolução do problema, este ainda é computacionalmente exigente, o que limita a sua implementação na prática. Este projecto testa o desempenho do algoritmo através de uma implementação utilizando *computação gráfica*, mais exactamente a plataforma de desenvolvimento NVIDIA CUDA. A plataforma tem sido, desde 2006, utilizada com sucesso na resolução de vários algoritmos computacionalmente exigentes, e o seu *hardware* e funcionamento é descrito em detalhe no Capítulo 3. A evolução e implementação do código, que é apoiado pela biblioteca CUBLAS, é descrito na Secção 3.4.4. O código utilizado assim como os resultados são descritos no Capítulo 4. O foco principal é na optimização de sinal para valores elevados de matriz (e por intermédio de maiores valores de amostragem/resultados), onde o *hardware* massivamente

paralelo é comparado com uma implementação de referência em CPU. Para esse efeito, foi desenvolvido um programa que executou uma série de testes destinados a calcular o desempenho e fiabilidade deste, e com os resultádos que se apresentam no Capítulo 5.

Capítulo 2

Compressed Sampling em Rádio-Frequência

2.1 Amostragem de Sinais Limitados em Frequência

A operação de amostragem de sinal (*sampling*) consiste na aquisição de uma série de valores discretos $x_r(d), d \in [0...N]; d, n \in \mathbb{N}$ a partir de um sinal contínuo no tempo $x(t), t \in [t_i, t_f]$ em determinados momentos de amostragem efectuados a uma taxa W no tempo. Pelo teorema de Nyquist-Shannon, pode-se estabelecer que se o sinal $x(t)$ consiste de informação que está limitada em frequência f_{max}, então a taxa de amostragem que permite a reconstrução do sinal original será $W = 2.f_{max}$, o que é denominado a *taxa de Nyquist*. Este é o valor mínimo para que seja possível reconstruir o sinal original através de uma soma de funções sinc 2.1.

$$x(t) = \sum_{n \in \mathbb{Z}} x_r(\frac{n}{W}) sinc(Wt - n) \tag{2.1}$$

2.2 Bases Teóricas do Compressed Sampling

As fundações base do processo de recuperação de sinais utilizando Compressed Sampling (CS) são baseados em dois fundamentos essenciais:

- *Esparsidade* - Deve ser possível representar o sinal através de um número limitado de valores não nulos. Por exemplo, um sinal discreto no tempo que possa ser representado por um sinal esparso na frequência através da transformada de Fourier.

- *Incoerência* - O sinal deve ser amostrado através de um método que gere resultados incoerentes, ou seja, que o resultado da amostragem não é efectuado através de um padrão regular de valores.

Se estas duas propriedades estiverem presentes, como iremos ver, é possível efectuar uma série de operações matemáticas para obter o sinal original, mesmo quando este foi obtido através de uma razão de aquisição muito inferior ao limite de Nyquist do sinal.

Importa aqui diferenciar este processo de outros resultados relativos à esparsidade, como é o caso de compressão *lossy* de imagem e áudio. Nestes casos um sinal é analisado para obter valores de esparsidade relevantes, procedendo-se então a criação, adaptativa, de uma versão reduzida do sinal com a mesma informação. Apesar de se tratar de um processo de compressão, o objectivo do CS é a obtenção de um sinal através de um método não-adaptativo, ou seja, onde a posição dos valores esparsos não tem relevância no processo de captura de sinal. De facto, prova-se [11] que a taxa de recuperação de CS é idêntica com valores aleatórios ou medidos.

Muitos sinais RF são esparsos ou compressíveis quando consideramos uma base Ψ apropriada. Um desmodulador aleatório pode ser utilizado para capturar sinais que consistam num sinal multifrequência com K frequências sinusoidais, ou seja, em que a base Ψ é o domínio da frequência. O resultado deste desmodulador é um sinal incoerente com uma razão de amostragem inferior ao que foi exigido.

2.3 Implementação do desmodulador aleatório

O dispositivo de desmodulação que tencionamos implementar foi definido em Tropp[15] e é descrito na Figura 2.1. Pretende-se obter a partir do sinal de entrada contínuo $x(t)$, uma lista de valores discretos $y(t_d)$. O sinal de entrada tem largura de banda máxima f_{max}, e a partir daí vamos definir $W = 2f_{max}$ e uma taxa de amostragem R, que nesta

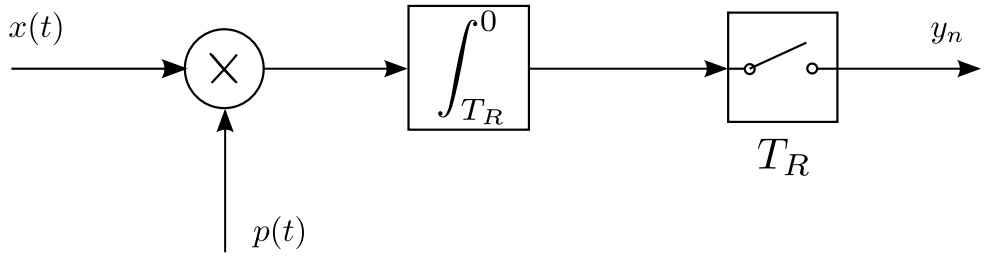

Figura 2.1: Esquema de um desmodulador aleatório

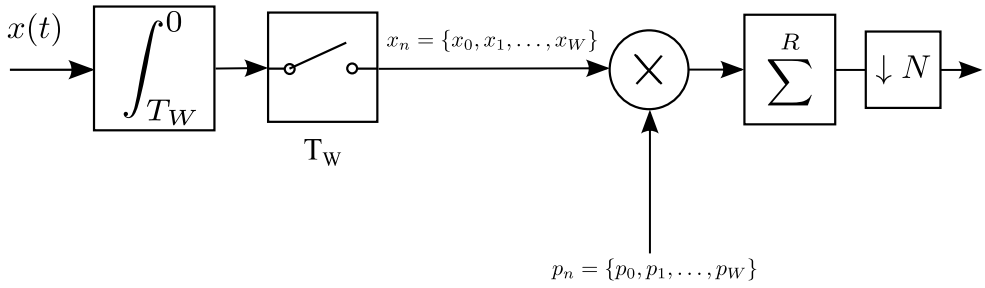

Figura 2.2: Esquema equivalente do desmodulador aleatório

Dado um sinal de entrada multi-frequência formado por uma soma de K sinusóides descritos por:

$$x(t) = \sum_{w \in \Omega} a_w e^{2\pi j \omega t} \ para\ t \in [0, 1] \tag{2.2}$$

em que Ω representa uma série de frequências de valor inteiro no subconjunto:

$$\Omega \subset \{0, \pm 1, \pm 2, \ldots, \pm(f_{max} - 1), f_{max}\} \tag{2.3}$$

e onde:

$$\{a_\omega : \omega \in \Omega\} \tag{2.4}$$

representa um conjunto desses valores. Interessa-nos o caso em que $K << W$, ou seja, que o sinal é K-esparso em termos de frequência. Será então possível aplicar CS a esse sinal incoerente no tempo. Para esse propósito um sinal aleatório $p(t)$ é multiplicado à entrada, posteriormente integrado a uma razão $T_R = 1/R$ e finalmente amostrada a uma taxa T_R.

O sinal $p(t)$ pode ser representado com uma série de pulsos rectagulares p_n com duração $T_W = 1/W$ e amplitude p_n:

$$p(t) = \sum_{k=0}^{W-1} p_k \Pi((t-k)W) \ \textit{para}\ p_k \in \{-1, 1\} \tag{2.5}$$

Isto torna o sinal p_n uma representação integral do valor de $p(t)$ durante um momento T_W. Embora o esquema da Figura 2.1 seja um modelo de *hardware* válido, é possível obter uma forma equivalente que permite obter um modelo discreto do problema ao qual se pode aplicar os algoritmos de CS. O objectivo é obter uma representação discreta x_n tal que este represente o valor de entrada amostrado a uma frequência $t_n = 1/W$ e em que cada elemento de x_n é o valor médio do sinal num período de tempo T_W como se pode observer na figure 2.2.

$$x_n = \int_{t_n}^{t_n + T_W} x(t)\, dt \tag{2.6}$$

Com x_n e p_n, a operação de modulação é representada pela multiplicação de ambos os valores discretos $x_n p_n$, somado em cada R blocos:

$$y_k = \sum_{n=t_k}^{t_r + W/R} x_n * p_n \, t_k = 0, (W/R), 2(W/R), \ldots, W \tag{2.7}$$

em que os valores discretos y_k representam a variação de valor de $y(t)$ em intervalos de 1/R. Esta representação compacta contém a informação do sinal, mas é necessário obter uma equivalência do sinal no seu domínio esparso, neste caso em frequência. Dada a equação

2.2, podemos representar 2.6 da seguinte maneira:

$$x_n = \int_{t_n}^{t_n+T_W} \sum_{\omega\in\Omega} a_\omega e^{j2\pi\omega t}\, dt \tag{2.8}$$

$$= \sum_{\omega\in\Omega} a_\omega \left[\frac{e^{2\pi j\omega T_W} - 1}{j2\pi\omega}\right] e^{-j2\pi\omega t_n} \tag{2.9}$$

onde se $\omega = 0$, o valor em parêntesis é igual a T_W. Utilizando o coeficiente, podemos escrever:

$$s_\omega = a_\omega \left[\frac{e^{j2\pi\omega T_R} - 1}{j2\pi\omega}\right] \tag{2.10}$$

e representar x_n na forma reduzida:

$$x_n = \sum_{\omega\in\Omega} s_\omega e^{-j2\pi\omega t_n} \;\; para\; n = 0, 1, \ldots, W-1 \tag{2.11}$$

Lembrando que nesta representação, Ω tem K elementos. A matriz permutada da transformada de Fourier $\mathbf{F}$ do sinal é estabelecida da seguinte maneira:

$$\mathbf{F} = \frac{1}{\sqrt{W}} \left[e^{-j2\pi n\omega'/W}\right]_{n,\omega'} \;\; em\; que \tag{2.12}$$

$$n = 0, 1, \ldots, W-1\; e \tag{2.13}$$

$$\omega' = 0, \pm 1, \ldots, \pm(W/2-1), \pm(W/2) \tag{2.14}$$

Dado que ω' inclui todos os valores de Ω, podemos definir que existe um vector $\mathbf{s}$, esparso, sob o qual:

$$\mathbf{x} = \mathbf{F}\mathbf{s} \tag{2.15}$$

Em que $\mathbf{x}$ é uma representação vectorial dos valores de x_n, e $\mathbf{s}$ terá K valores não-nulos positivos, nos casos em que $\omega' \in \Omega$. A partir do momento em que $x(t)$ tem uma representação discreta em $\mathbf{x}$, podemos tratar o problema da modulação através de uma operação matricial. Sejam $p_n = \{p_0, \ldots, p_W - 1\}$ os W valores discretos aleatórios que geram $p(t)$, então seja uma matriz $\mathbf{P}$ tal que:

$$\mathbf{P} = \begin{bmatrix} p_0 & & & & \\ & p_1 & & & \\ & & \dots & & \\ & & & p_{W-2} & \\ & & & & p_{W-1} \end{bmatrix} \tag{2.16}$$

Como já definimos anteriormente em 2.7, para obter os valores discretos y_k, é necessário multiplicar os valores correspondentes a cada valor discreto, somados num conjunto de R entradas representadas por um vector $\mathbf{y}$. Se considerarmos um valor fixo de W amostras no total, existem W/R valores em $\mathbf{y}$. Esta operação pode ser representada por uma matriz H de dimensão $W \times R$, onde cada linha r apresenta W/R valores unitários a começar na posição $r.(W/R)$. Num exemplo simples, onde $R = 3$ e $W = 12$, $\mathbf{H}$ será:

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & & & & & & & & \\ & & & & 1 & 1 & 1 & 1 & & & & \\ & & & & & & & & 1 & 1 & 1 & 1 \end{bmatrix} \tag{2.17}$$

Se calcularmos a matriz $\mathbf{M} = \mathbf{HP}$ torna-se óbvio que esta permite converter o vector de entrada discreta $\mathbf{x}$ nos valores discretos da saída comprimida y, pois, baseando-nos no último exemplo:

$$\mathbf{Mx} = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 & & & & & & & & \\ & & & & p_4 & p_5 & p_6 & p_7 & & & & \\ & & & & & & & & p_8 & p_9 & p_{10} & p_{11} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_{10} \\ x_{11} \end{bmatrix} = \tag{2.18}$$

$$= \begin{bmatrix} x_0.p_0 + x_1.p_1 + x_2.p_2 + x_3.p_3 \\ x_4.p_4 + x_5.p_5 + x_6.p_6 + x_7.p_7 \\ x_8.p_8 + x_9.p_9 + x_{10}.p_{10} + x_{11}.p_{11} \end{bmatrix} = \mathbf{y} \tag{2.19}$$

Se desejarmos efectuar a operação inversa, ou seja, obter o valor de $\mathbf{x}$ a partir de um valor amostrado $\mathbf{y}$, podemos utilizar 2.15 e multiplicar esta nova matriz:

$$\mathbf{x} = \mathbf{Fs} \Rightarrow \mathbf{Mx} = \mathbf{MFs} \Rightarrow \mathbf{y} = \mathbf{MFs} \tag{2.20}$$

Em que $\Phi = \mathbf{MF}$ é denominada a *matriz de desmodulação aleatória*. Apesar de ser uma solução elegante, não é um sistema de resolução trivial. Φ trata-se de uma matriz não-quadrada que resolve a equação num sistema sobre-determinado. A solução passa então pelo problema de minimização, neste caso:

$$\hat{s} = argmin \left\| v \right\|_0 \text{ que resolve } \Phi v = y \tag{2.21}$$

A resolução deste problema requer uma pesquisa exaustiva para todas as combinações possíveis do sinal v para cada valor de esparsidade. No entanto alguns autores demonstraram que se pode obter a mesma solução se se resolver o problema

$$\hat{s} = argmin \left\| v \right\|_1 \text{ que resolve } \Phi v = y \tag{2.22}$$

que permite utilizar algoritmos de programação linear para encontrar a solução esparsa desde que a matriz Φ obedeça a determinadas condições [6, 16, 1]. Contudo estes algoritmos baseados na programação linear – também conhecidos por *Basis Pursuit* são demasiado pesados do ponto de vista computacional para poderem ser utilizados em aplicações de tempo real. Em alternativa podemos utilizar algoritmos do tipo "greedy" que conseguem um desempenho semelhante mas com um menor peso computacional. Na secção seguinte serão apresentados dois destes algoritmos, o *Matching Pursuit* e o *Orthogonal Matching Pursuit*.

Para demonstrar a operação deste sistema, foi criada uma implementação em MATLAB do descodificador aleatório, presente no Anexo A.1. A Figura 2.3 exibe o detalhe de um sinal processado por esta ferramenta, em que $W = 1000$, $R = 200$ e o sinal de entrada tem esparsidade na frequência de 50. O sinal é amostrado em 200 amostras e posteriormente recuperado integralmente.

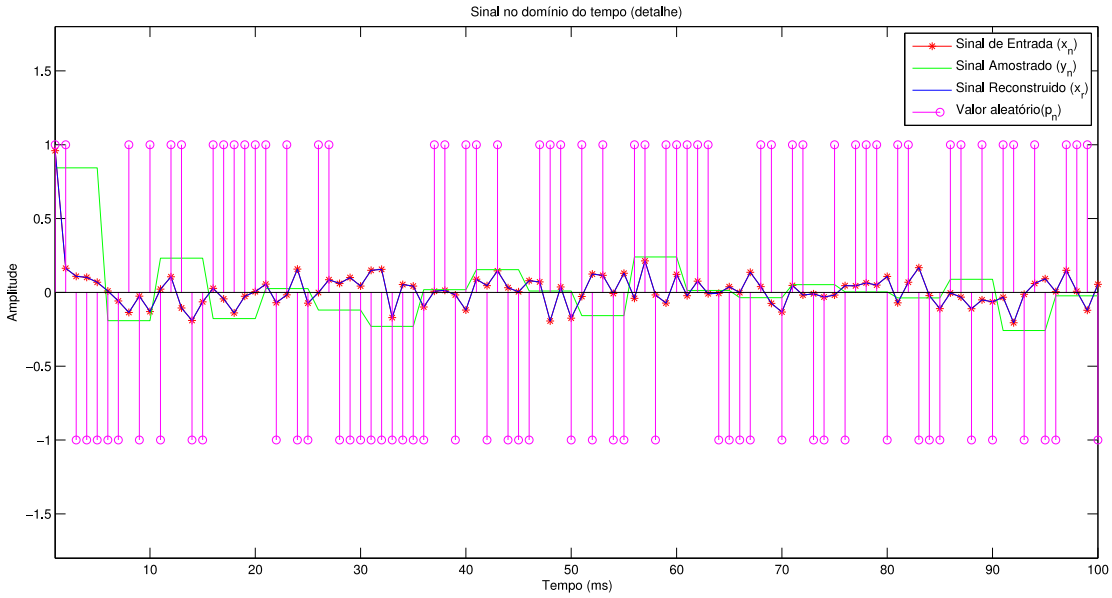

Figura 2.3: Detalhe de um sinal esparso recuperado (domínio dos tempos)

2.4 Resolução do problema esparso: Matching Pursuit

O processo de recuperação ou desmodulação de um sinal em CS depende da capacidade de resolução do problema esparso referido na Equação 2.21. Uma das variantes mais populares é intitulado *Matching Pursuit* (MP)[10], que permite a recuperação do sinal através de uma série de aproximações do sinal original. Embora o algoritmo em si esteja descrito em 4.1, aqui apresenta-se uma explicação gráfica da sua operação. Seja um vector de entrada s em $\mathbb{R}^2$ tal que:

$$s = (s_j, s_k) \, \|s\| = 1 \tag{2.23}$$

e uma matriz Φ tal que:

$$\Phi = \begin{bmatrix} \phi_{11} & \phi_{12} \\ \phi_{21} & \phi_{22} \end{bmatrix} = \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix} \; \phi_1, \phi_2 \in \mathbb{R}^2, \|\phi_1\| = \|\phi_2\| = 1 \tag{2.24}$$

Uma representação deste valor é calculado geométricamente na Figura 2.4(a). Note-se que neste caso a matriz tem tamanho idêntico ao sinal de entrada e de amostragem, mas

que isso não altera os passos essenciais do algoritmo. Neste caso vamos definir **y** como um vector de amostragem, tal que:

$$y = \Phi.\mathbf{s} = (\phi_{11}.s_j + \phi_{12}.s_k, \phi_{21}.s_j + \phi_{22}.s_k) = s_j.\phi_1 + s_k.\phi_2 \tag{2.25}$$

Este valor é igualmente calculado e representado na Figura 2.4(b). Seja **s'** um vector nulo que armazena a solução. Neste caso, o método de Matching Pursuit pode então ser representado pelos seguintes passos:

1. Um vector ϕ_k é determinado que apresente a maior projecção de r de cada vector linha de Φ, ou seja tal que $\|\langle y, \phi_k \rangle\|$ seja máximo.
2. O valor do produto interno $\|\langle \mathbf{y}, \phi_k \rangle\|$ é adicionado à dimensão k do vector solução x'.
3. Uma projecção do sinal r é calculada através de $d_k.\|\langle \mathbf{y}, d_k \rangle\|$.
4. Os passos 1 a 3 são repetidos até a norma de **y** ser nula.

A Figuras 2.4(c) e (d) ilustra o algoritmo descrito. O primeiro passo resulta em caso $\phi_k = \phi_2$, e logo $t = (0, \|\langle s, \phi_2 \rangle\|)$. y_0 representa o novo valor de **y** para o segundo ciclo, que é caulclado com vista a obter y_1. Como esperado, o valor de **s'** vai tender para o valor original s a medida que a norma de y diminui. Este processo é computacionalmente simples, mas prova-se [10] que é necessário um grande número de iterações para obter um valor satisfatório.

As limitações do algoritmo de Matching Pursuit podem ser evitadas através de uma versão do algoritmo definida por Orthogonal Matching Pursuit (OMP)[11]. Este algoritmo, descrito na Secção 4.1, calcula um valor de t que é directamente ortogonal aos vectores d_k em cada iteração. Este processo, embora computacionalmente mais exigente, resulta num número limitado de iterações até ao seu limite, mais exactamente, e para um valor N-esparso serão necessárias apenas N iterações do algoritmo.

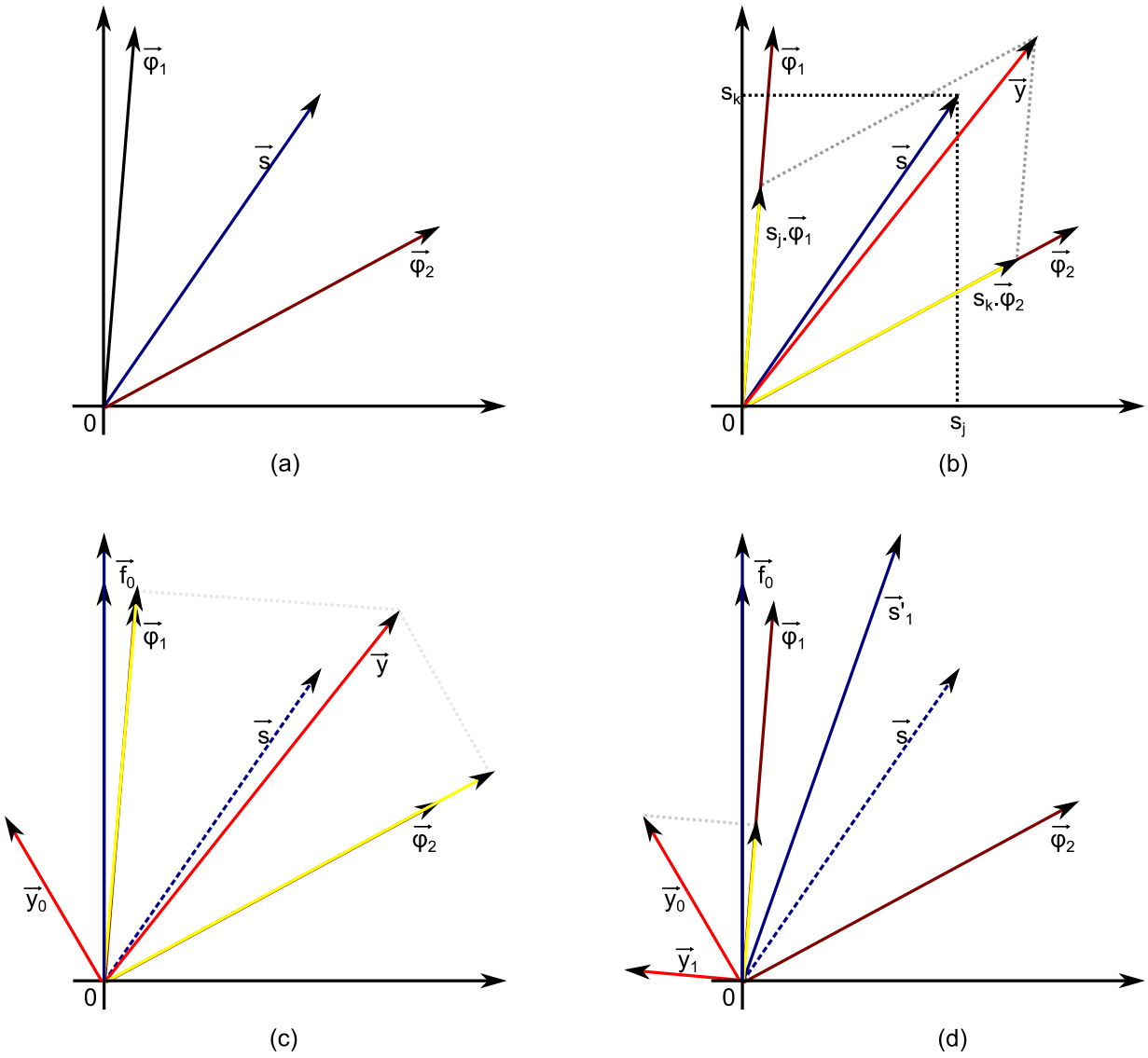

Figura 2.4: Demonstração gráfica do algoritmo de Matching Pursuit. (a) O vector de entrada e dicionário. (b) Cálculo do vector amostrado $\mathbf{y}$. (c) Primeiro passo de MP (d) Segundo passo de MP.

Capítulo 3

CUDA

Dado que pretendemos recuperar do sinal obtido pelo descodificador aleatório através do algoritmo de Matching Pursuit, uma implementação prática implica a capacidade e efectuar uma grande quantidade de cálculos em tempo real. A tecnologia gráfica actual que utiliza Unidades Gráficas de Processamento (Graphics Processing Units ou GPUs) foi desenvolvida para resolver problemas semelhantes, ou seja operações que necessitam de ser efectuadas num curto espaço de tempo e que sejam baseadas em matrizes. Importa então estudar e verificar se existem vantagens ao implementar o algoritmo em plataformas baseadas em GPUs.

3.1 GPU e Computação Paralela

O conceito de utilização de processadores paralelos programáveis em *hardware* de consumo surgiu há pouco mais de uma década. A sua aplicação começou por um desejo de aumentar o desempenho gráfico do traçado de gráficos 3D em tempo real. Numa aplicação normal, a transformação de um objecto 3D (definido por uma série de recursos, incluindo uma lista de vértices e texturas), necessita de ser correctamente alterado para corresponder à vista desse objecto por uma "camera" virtual. Adicionalmente, efeitos como o cálculo de iluminação e oclusão são factores importantes para a criação de um cenário realista.

Essas operações normalmente implicam o tratamento de grandes quantidades de informação, todas estruturadas de forma semelhante e às quais são sujeitas as mesmas transformações. Esperava-se então que a tecnologia de processamento seguisse a lei de Moore e oferecesse CPUs capazes de efectuar estas operações complexas, mas isso provou-se ineficaz a partir da última década, em parte devido a limitações físicas do processo de fabrico. Isto levou os fabricantes de placas gráficas a adicionar lógica adicional para o cálculo dessas operações matemáticas e a adoptar arquitecturas paralelas nos seus CPUs para o meracado doméstico - por exemplo, todos os CPUs de arquitectura x86 de consumo disponíveis actualmente pela Intel e AMD incluem pelo menos dois *cores* dentro de cada unidade lógica.

Transform and Lighting (T&L) foi a primeira estrutura universalmente adoptada em GPUs para o processamento de operações matemáticas. Consistia numa API de acesso que efectuava operações essenciais de transformação de matrizes utilizando *hardware* dedicado, mas capazes apenas de operações atómicas pré-definidas que não podiam ser alteradas no processo de criação da imagem. O passo seguinte foi a criação de *shaders*, em que elementos individuais de uma imagem 3D (vértices, texturas) são utilizados como valores para efectuar operações elementares. Estes foram divididos em *vertex shaders* e *pixel shaders*, em que cada tipo de *shader* apresentava uma estrutura e operações optimizadas para o tipo de dados que processavam.

O interesse em aproveitar essa nova capacidade de programação em tarefas de processamento genéricas (normalmente denominado por General Purpose GPU ou GPGPU) surgiu quase simultaneamente. Em 2003, Krüger[9] apresentava as potencialidades do cálculo de álgebra linear em GPU, utilizando texturas como entrada e saída da informação e um ganho de aproximadamente 15 vezes em relação a uma implementação equivalente em CPU. No entanto, nem a linguagem nem o API de acesso ao *hardware* tinham sido concebidos para este propósito, o que tornava a implementação de *software* fora do campo experimental relativamente complexo.

Cabe à NVIDIA lançar em 2006 a API de acesso genérico Compute Unified Device Architecture (CUDA) para a plataforma gráfica G80, que apresentava uma nova arquitectura

que apresentava *shaders unificados*, presente em placas como a GeForce 8800 Ultra. Ao contrário das implementações anteriores de *shaders*, estes elementos não estavam divididos por funcionalidade e podiam aceder a vários tipos de memória indescriminadamente. A nova API, manifestada através de uma extensão de ANSI C, também contribuía para código mais limpo, fácil de seguir e que, não estando ligada directamente a uma linguagem gráfica como o OpenGL, facilmente se integrava em projectos já existentes.

Actualmente, a NVIDIA já providencia unidades GPGPU (denominadas Tesla) que não incluem interface de vídeo. Estas placas podem ser montadas em *clusters* de processamento paralelo, ou montadas numa *workstation*, onde são utilizadas nos campos da investigação e simulação.

3.2 Arquitectura

As diferenças de arquitectura entre um CPU e um GPU são um elemento importante para compreender as vantagens específicas de cada. A arquitectura para CPUs mais comum é a de von Newmann, onde uma série de instruções são executadas sequencialmente. A vantagem principal deste esquema consiste em ser fácil de programar, dado que uma sequência lógica de eventos pode ser seguida. Considerando um esquema de blocos (Figura 3.1) que directamente relaciona a quantidade de espaço em silício utilizado e a função que cada elemento executa, um CPU é dominado por uma grande quantidade de memória de alta velocidade, um elemento de controlo de operação e um número limitado de Unidades Lógicas e Aritméticas (Arithmetic and Logic Units ou ALUs). A memória rápida, assim como a unidade de controlo, permite optimizar uma linguagem focada em programação linear.

A arquitectura de um GPU toma uma posição diferente em termos de organização interna. No diagrama de blocos equivalente (Figura 3.2), apenas uma pequena quantidade de *cache* e sistemas de controlo é colocado em paralelo com uma grande quantidade de ALUs. Cada um destes blocos é então capaz de processar um número elevado de valores simultaneamente, o que lhe dá uma vantagem significativa em várias operações computacionais

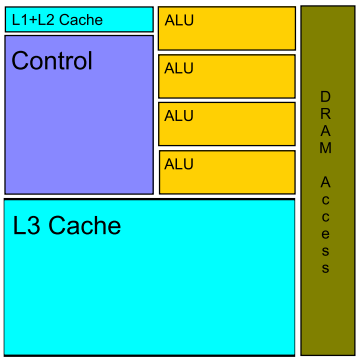

Figura 3.1: Arquitectura de um CPU clássico (em relação à ocupação de silício)

quando comparado com um CPU de velocidade igual ou superior.

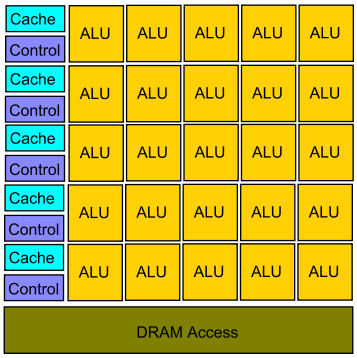

Figura 3.2: Arquitectura de um GPU com *shaders* unificados(em relação à ocupação de silício)

No entanto, esta solução inclui algumas desvantagens, pois as unidades de controlo são comparativamente muito mais simples e controlam directamente uma grande quantidade de ALUs - isto implica que o código deve explorar o máximo de lógica matemática simultânea, e evitar ciclos de decisão que fragmentem o algoritmo. Logo, o aumento teórico de prestação de um GPU está directamente ligado com a capacidade de detectar e implementar *software* que explore a fundo as características do *hardware*. Tomando em conta uma visão mais precisa da arquitectura unificada corrente no *hardware* da NVIDIA (Figura 3.3), encontramos vários elementos individuais:

- **Stream Processors(SP)**: Cada SP consiste numa unidade de cálculo matemático,

e em termos de *hardware* trata-se de um *core* ou núcleo de processamento que acede directamente à memória partilhada.

- **Stream Multiprocessors(SM)**: Por sua vez, um número de SPs são reunidos num SM. Um SM inclui uma Instruction Unit (IU) que processa as instruções e distribui os dados pelos cores, assim como a memória partilhada por entre SPs internos. A NVIDIA classifica o processamento interno de um SM como SIMT (Single Instruction, Multiple Threads).

- **Thread Processing Cluster(TPC)**: Consecutivamente, vários SM estão organizados dentro de um TPC. Um TPC define a ordem e execução dos SM, e uma *cache L1* (ou seja, uma pequena quantidade de memória rápida e estática) que armazena os valores a ser processados. Qualquer unidade SM interna ao TPC pode aceder a este memória através de múltiplos *bus* de dados, o que contribui em grande parte para a eficácia dos algoritmos a ser executados. A classificação do processamento dentro de um TPC é MIMD(Multiple Instructions, Multiple Data).

- **Thread Scheduler**: O *Thread Scheduler* gere as *threads* existentes em cada TPC e resolve problemas de latência. Por exemplo, threads que esperam acesso à memória externa podem ser colocadas em espera sem perca de desempenho enquanto se efectua um *context switching* para uma *thread* que não necessite dessa operação.

- **Texture Memory**: A memória existente dentro de cada dispositivo individual. Esta está particionada em vários elementos com o seu próprio *bus* de dados, de maneira a maximizar a largura de banda necessária.

- **Host Memory Access**: Para acesso ao sistema que aloja o dispositivo CUDA, uma série de operações elementares são possíveis para transferência de dados de e para a memória global. Adicionalmente, é possível converter alguma da memória global em memória *pinned* que possibilita o acesso directo a esta pela unidade, embora a custo de uma menor largura de banda.

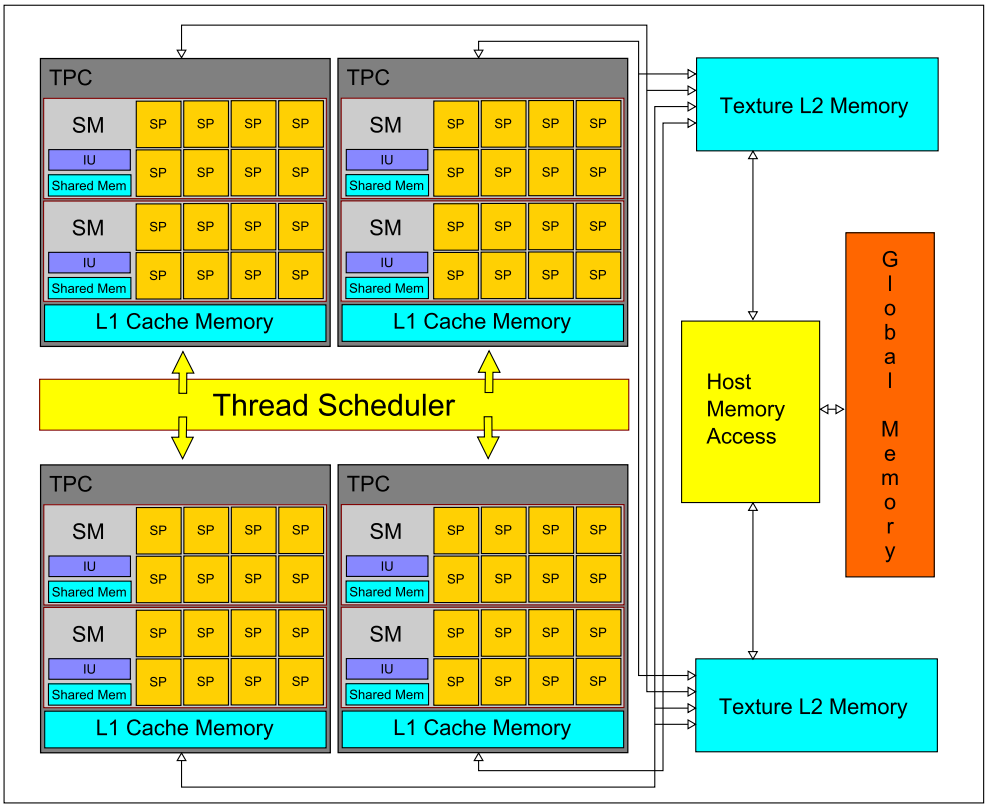

Figura 3.3: Arquitectura global de um GPU NVIDIA em relação às unidades disponíveis em CUDA

Cada geração da arquitectura CUDA tem valores diferentes de SPs por cada unidade SM e de SMs por cada TPC. Por conveniência, os valores para cada variante mais significativa estão listados na Tabela 3.1.

Arquitectura	SMs por SP	SPs por TPC
G80/G90	8	2
G200	8	3
GF100	32	4
GF104/6/8	48	4
GF110	32	4

Tabela 3.1: Organização dos elementos de processamento por arquitectura NVIDIA

3.2.1 Compute Capability e Warps

Para permitir a evolução da plataforma, a NVIDIA atribui a cada GPU que suporte CUDA um valor de *Compute Capability* correspondente à disponibilidade de determinadas funções do *hardware*. Observando a Tabela 3.2 é possível verificar que se trata de um processo de melhoria com a introdução de novas características, em que as novas revisões mantêm as mesmas capacidades das versões anteriores. Isto significa que, se se desejar efectuar cálculos em vírgula flutuante com dimensão *double* utilizando *hardware* CUDA, esta será possível a partir de *hardware* com Compute Capability 1.3 e superior, mas o *software* não irá executar em *hardware* com Compute Capability 1.0 ou 1.1. É importante verificar se o *hardware* que se deseja utilizar suporta as características desejadas. O equipamento utilizado para a implementação em GPU do algoritmo apresentava Compute Capability 1.3, e logo apenas as funcionalidades presentes nesse nível de *hardware* serão aqui mencionadas.

Um factor importante associado a ao nível de Compute Capability é o conceito de *warp*. Um *warp* refere-se ao número de tarefas que podem ser executadas em simultâneo por um SM, assim como a máxima quantidade de instruções simultâneas a serem executadas dentro de cada SM. Estas tarefas são decididas internamente pelo driver CUDA ao executar o código, e iniciadas através do *Thread Sheduler*.

No caso da Compute Capability 1.3, um warp indica a execução simultânea dos 8 *cores* em cada SM, entregando uma instrução de cada vez (num total de 32 wraps a ocorrerem simultaneamente). A execução de um *warp*, no entanto, também estabelece as regras de acesso à memória, e no caso do *hardware* em questão este pode ser divido em *half-warps*, ou seja, que cada *warp* dentro de um SM pode executar dois acessos à memória global. No entanto, o CUDA é igualmente capaz de interpretar acessos consecutivos à memória de uma *kernel*, e subsequentemente *agregar* (ou seja, agrupar uma série de acessos simultâneos) o acesso à memória. Apesar da decisão de um acesso simultâneo ser controlada pelo *driver* CUDA e o *Thread Scheduler* o código necessita de ser devidamente organizado, ou seja, que cada *core* possa aceder à memória de uma maneira conjunta para ser devidamente

interpretado como um acesso agregado.

Operações/Características	Compute Capability				
	1.0	1.1	1.2	1.3	2.x
Grelhas *threads* tridimensionais					X
Operações atómicas inteiras de 32-bits com memória global			X	X	X
Operações atómicas inteiras de 64-bits com memória global			X	X	X
Operações atómicas inteiras de 32-bits com memória partilhada			X	X	X
Operação de eleição de *warp*			X	X	X
Operações de vírgula flutuante com dupla precisão				X	X
Operações atómicas de adição em vírgula fultuante em 32-bits em memória global ou partilhada				X	X
_ballot()					X
_threadfence_system()					X
_syncthreads_count(), _syncthreads_and(), _syncthreads_or()					X
Funções de cálculo de superfície					X

Tabela 3.2: Lista de caracteristicas suportadas por Compute Capability

3.3 Hardware

Actualmente todo o *hardware* disponível e futuro da NVIDIA suporta CUDA, sendo este dividido em *hardware* de consumo, *hardware* para *workstations*, e soluções para servidores *rack-mounted*. Esta secção detalha o *hardware* anterior e as suas capacidades, assim como as linhas gerais que levaram ao desenvolvimento do dispositivo.

3.3.1 GeForce : Desktop e Portátil

O *hardware* de consumo da NVIDIA com suporte CUDA iniciou-se com o lançamento da GeForce 8800 e da arquitectura G80 de processadores, que utiliza o processo de fabrico de 80nm. Lançada em 2007, a GeForce 8800 contém o máximo de Stream Processors da arquitectura G80 de consumo (128), e está disponível com 768Mb de Memória GDDR3 na sua versão Ultra3.4 com uma largura de *bus* de 376 bits. Esta arquitectura foi depois retomada com uma tecnologia de fabrico de 65nm denominada série G90, que foi utilizada nas versões GS, GT e GTS desta placa, assim como na produção de uma versão 8800M dedicada a portáteis.

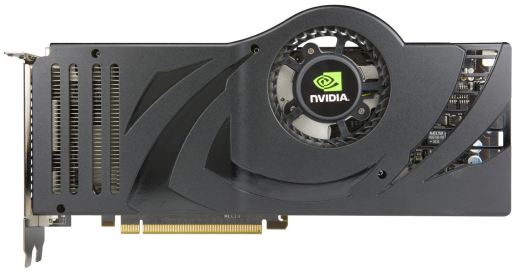

Figura 3.4: NVIDIA GeForce 8800 Ultra

O G90 foi depois utilizado na fabricação da GeForce Série 9, que aumenta o número de SM para 256 na GeForce 9800 GTX e torna standard o uso do interface PCIe 2.0 para a comunicação com o *host*. O menor consumo de energia conseguido com a utilização da tecnologia de 65nm permite uma maior velocidade de relógio no GPU, assim como o aumento aumento de *cores*, mas apresenta a mesma Compute Cabability 1.1 da geração anterior. A arquitectura G90 foi também utilizada para a produção da GeForce Série 100, com poucas ou nenhumas alterações.

A primeira completa renovação do *hardware* CUDA veio com a arquitectura GT200 e a GeForce 280GTX (Figura 3.5). Lançada em 2008, esta arquitectura disponibiliza até 240 SM para processamento e uma aumento da largura de bus de memória para 512 bits. Também é aqui feita a primeira alteração na distribuição dos SM (ver Tabela 3.1), assim como o aumento do tamanho dos registos internos dos SM (de *float* para *double*), o

que duplica a capacidade de processamento teórico da unidade. Estas foram igualmente lançadas sob a marca GeForce Série 300.

Figura 3.5: NVIDIA GeForce GTX 280

Devido à necessidade de melhorar o desempenho para os novos APIs 3D, a NVIDIA efectuou uma completa renovação da sua oferta de placas gráficas em 2010. Originalmente com nome de código Fermi, esta foi posteriormente denominada GF100 e é vendida sob a forma da GeForce Séries 400/500. Devido a problemas de fabricação, os GPUs da mesma família variam um pouco em capacidade individual. As placas com arquitectura GF100 apenas utilizam 32 dos 48 SMs disponíveis por SP, uma limitação que não existe na GF104/6/8. Simultaneamente, a utilização de tecnologia de 40nm para uma versão actualizada do processador, intitulada GF110, permitiu a utilização do número máximo de SPs disponíveis dentro da arquitectura - 16 SPs (com 32 SM cada) que eleva o número máximo de SPs na arquitectura a 512, quantidade disponível na GeForce GTX 580 (Figura 3.6).

Figura 3.6: NVIDIA GeForce GTX 580

3.3.2 Equipamento utilizado

A plataforma GPU de desenvolvimento consistiu no *NVIDIA Tesla S1070 GPU Computing Server*. Este sistema reúne um total de quatro cores GPU baseados na série GF200, cada uma com 4GB e capazes de processar até um máximo de 933Giga Floating Point Operations Per Second(GFLOPS) para um total teórico de 4147.2 GFLOPS[3], estando ligadas aos pares utilizando um controlador da NVIDIA que optimiza o uso da largura de banda. Com o objectivo de estabecer um sistema que aproveitasse ao máximo a capacidade da Tesla S1070, foi decidido a instalação de um servidor dedicado para esta.

Em particular, para os 4 elementos estarem acessíveis e com o máximo de largura de banda, era exigido um PC que suportasse dois slots PCI-Express 2.0 16x. Apesar de alternativas baseadas em controladores externos estarem disponíveis, a escolha recaiu sobre a motherboard Asus P6T WS Professional, baseada no controlador Intel X58 Express que suporta tal configuração nativamente. Para correr a plataforma Linux, foi escolhido um processador Intel Core i7-950 com uma velocidade de relógio de 3.3Ghz e marcado como capaz de efectuar até 50GFLOPS[2].

3.4 NVIDIA CUDA e CUBLAS

Como já foi aqui descrito, o processo de programação de um GPU é crítico para o desempenho deste, e implica em muitos casos a aplicação de técnicas de programação que explorem o *hardware* ao máximo. Ao longo dos últimos anos, vários esforços têm sido feitos para a criação de um interface com o *hardware* que simplifique este acesso, dos quais o CUDA é o mais popular. Com execução exclusiva no *hardware* da NVIDIA, a API CUDA consiste numa extensão da linguagem C e C++ que permite utilizar ao máximo as capacidades de programação *multithreaded*. Nesta secção é descrita a instalação, configuração e execução de código CUDA e da biblioteca CUBLAS incluída.

3.4.1 Requisitos

Os requisitos iniciais para a instalação de uma plataforma CUDA em Linux são os seguintes:

- *Hardware* NVIDIA com suporte CUDA: Além do equipamento referido na secção 3.3, uma lista actualizada do *hardware* que suporta CUDA está disponível no site do fabricante[5];

- Sistema operativo Linux de 32 ou 64 bit: O tipo de sistema operativo Linux em execução pode ser determinado usando o comando:

  ```
  uname -m && cat /etc/*release
  ```

  O qual retorna uma série de valores tais como:

  ```
  x86_64
  DISTRIB_ID=Ubuntu
  DISTRIB_RELEASE=10.10
  DISTRIB_CODENAME=maverick
  DISTRIB_DESCRIPTION="Ubuntu 10.10"
  ```

  com a linha inicial a indicar que se trata de um sistema operativo de 64-bit, (no caso em que se trata de um S.O. de 32-bits, retornará i386);

- Uma versão do gcc instalado, verificado via execução de:

  ```
  gcc --version
  ```

  que deverá retornar a versão instalada actual:

  ```
  gcc (Ubuntu/Linaro 4.4.4-14ubuntu5) 4.4.5
  Copyright (C) 2010 Free Software Foundation, Inc.
  This is free software; see the source for copying conditions.  There is NO
  warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
  ```

No caso de não estar disponível, é necessário verificar ou instalar o pacote de ferramentas de desenvolvimento da versão de Linux em questão.

Todo o *software* necessário pode então ser descarregado no site do fabricante[4], nomeadamente o *Linux Developer Driver*, o *CUDA Toolkit* para a respectiva versão do SO, e o *CUDA Computing SDK* com o restante *software*. As secções seguintes descrevem o processo de instalação e execução de código neste ambiente. No caso em que todo o *software* já foi instalado (por exemplo, se se acede a um servidor Tesla por acesso remoto), a Secção 3.4.3 define os passos para adicionar à conta do utilizador os elementos essenciais para utilizar o SDK CUDA.

3.4.2 Instalação do *Developer Driver*

Para efectuar a instalação do driver, é necessário desactivar o interface gráfico do computador (GUI) se este estiver activado. Isto pode ser efectuado na maior parte das instalações de Linux através do comando **Control+Alt+Backspace**. Dirigindo-se para a directoria onde o *driver* foi descarregado, basta executar o pacote respectivo como *superutilizador*, por exemplo:

```
sudo devdriver_4.0_linux_64_270.41.19.run
```

Após a instalação é possível verificar se os *drivers* foram correctamente instalados via:

```
cat /proc/driver/nvidia/version
```

Os valores devem coincidir com o *driver* instalado, ou seja, na versão utilizada anteriormente:

```
NVRM version: NVIDIA UNIX x86_64 Kernel Module  270.41.19  Mon May
16 23:32:08 PDT 2011
GCC version:  gcc version 4.4.5 (Ubuntu/Linaro 4.4.4-14ubuntu5)
```

No caso do sistema não apresentar um GUI ou este não utilizar uma placa NVIDIA, pode ser necessário um passo adicional, nomeadamente a criação dos ficheiros /dev/nvidia* com as permissões correctas:

```
crw-rw-rw- 1 root root 195,   0 2011-08-26 14:30 /dev/nvidia0
crw-rw-rw- 1 root root 195,   1 2011-08-26 14:30 /dev/nvidia1
crw-rw-rw- 1 root root 195,   2 2011-08-26 14:30 /dev/nvidia2
crw-rw-rw- 1 root root 195,   3 2011-08-26 14:30 /dev/nvidia3
crw-rw-rw- 1 root root 195, 255 2011-08-26 14:30 /dev/nvidiactl
```

em que a numeração corresponde ao número de GPUs disponíveis no sistema. Este código, retirado do guia de instalação, pode ser adicionado aos scripts de inicialização do sistema para efectuar o processo automaticamente:

```
#!/bin/bash
/sbin/modprobe nvidia
if [ "$?" -eq 0 ]; then
  # Count the number of NVIDIA controllers found.
NVDEVS=‘lspci | grep -i NVIDIA‘
  N3D=‘echo "$NVDEVS" | grep "3D controller" | wc -l‘
  NVGA=‘echo "$NVDEVS" | grep "VGA compatible controller" | wc -l‘
  N=‘expr $N3D + $NVGA - 1‘
  for i in ‘seq 0 $N‘; do
    mknod -m 666 /dev/nvidia$i c 195 $i
  done
mknod -m 666 /dev/nvidiactl c 195 255
else
  exit 1
fi
```

No caso do sistema incluir um GUI, o reinício do GUI via, por exemplo:

```
startx
```

deverá inicializar correctamente os dispositivos.

3.4.3 Instalação do CUDA Toolkit e GPU Computing SDK

Após a instalação com sucesso do Developer Driver, a instalação do CUDA Toolkit passa igualmente pela execução do ficheiro descarregado como superutilizador:

```
sudo cudatoolkit_4.0.17_linux_64_ubuntu10.10.run
```

Que instala todo o *software* necessário na pasta **/usr/local/cuda**. Após este processo, ou no caso de este já ter sido efectuado por um administrador, é possível agora configurar a conta do utilizador. O primeiro passo é adicionar a localização do CUDA Toolkit e respectivas bibliotecas ao ambiente, normalmente adicionando as seguintes linhas ao ficheiro **/.bashrc**:

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PAT
```

A instalação do GPU Computing SDK passa finalmente pela instalação do SDK na zona do utilizador, normalmente pela execução directa do código:

```
cudatoolkit_4.0.17_linux_64_ubuntu10.10.run
```

Que permitirá instalar uma copia do SDK na directoria **/NVIDIA_GPU_Computing_SDK**. É possível então compilar alguns dos exemplos existentes na biblioteca:

```
cd ~/NVIDIA_GPU_Computing_SDK/C/src/bandwidthTest
make
cd ~/NVIDIA_GPU_Computing_SDK/C/src/deviceInfo
make
```

Estes exemplos devem compilar com sucesso e podem ser executados a partir da localização **/NVIDIA_GPU_Computing_SDK/C/bin/linux/release/**.

3.4.4 Implementação de código CUDA

A criação de código CUDA é, como descrito anteriormente, uma extensão da linguagem C, que permite a definição de *kernels*. *Kernels* são subprogramas que correm dentro do GPU, normalmente denominado por *device*, e que são controlados por um computador ou *host*. A extensão `__global__` define uma função como um *Kernel*, e a sua invocação é criada através da *execution configuration* `<<< ... >>>`, o qual permite definir o número de *threads* executadas em simultâneo.

Para demonstrar o estilo de programação de CUDA, apresenta-se em seguida um exemplo simples, neste caso a soma de dois vectores de dimensão arbitrária. Em linguagem C, trata-se de uma operação simples:

```
void vecSum(float* a, float* b, float* c, int size)
{
    for (int i = 0; i < size; i++) c[i] = a[i] + b[i];
}
```

A implementação do algoritmo em CUDA normalmente toma a forma de duas funções, nomeadamente uma função de configuração da *kernel*, e a *kernel* propriamente dita. A função de configuração também pode efectuar a cópia dos dados a serem processados para a memória do *device* e posteriormente a sua recuperação, utilizando equivalentes das funções *standard* do C MALLOC() e MEMCPY() fornecidas pela API CUDA:

```
#define _TWIDTH  512
#include <cuda.h>


__global__ void runSumV(float* a, float* b, float* c, int size)
{
    // Each thread recieves individual threadIdx.x and blockIdx.x sets
    float result, i;  // local variables are stored in SPU memory
```

```
    i = threadIdx.x+_TWIDTH*blockIdx.x; // Calculate position to fetch
                                        // from array based on thread vars
    if(i < size){
        result = a[i] + b[i];           // Vector added in SPU memory
        c[i] = result;                  // Copy it to global GPU memory
    }
}

void vecSumGPU(float* a, float* b, float* c, int size)
{
    int block;
    float* gpu_a;  // holds operation data in GPU
    float* gpu_b;
    float* gpu_c;

    // the following functions copy the A, B and C vectors to GPU
    cudaMalloc((void **) &gpu_a, size*sizeof(float));
    cudaMemcpy(gpu_a, a, size*sizeof(float), cudaMemcpyHostToDevice);
    cudaMalloc((void **) &gpu_b, size*sizeof(float));
    cudaMemcpy(gpu_b, b, size*sizeof(float), cudaMemcpyHostToDevice);
    cudaMalloc((void **) &gpu_c, size*sizeof(float));

    block = size / _TWIDTH;  // calculate amount of blocks required

    dim3 dimBlock(_TWIDTH, 1, 1);  // this example is 1-dimensional
    dim3 dimGrid((block+1), 1, 1); // for Block and Grid units

    // The following function call is CUDA-compiler specific
    runSumV <<< dimGrid, dimBlock >>> (gpu_a, gpu_b, gpu_c, size);
```

```
    // Only the solution is copied back to the Host/CPU memorty
    cudaMemcpy(c, gpu_c, size*sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(gpu_a);
    cudaFree(gpu_b);
    cudaFree(gpu_c);
}
```

Neste exemplo, pode-se considerar que cada *thread* (ou cada invocação da *kernel* `vecSumV`) efectua uma operação elementar que consiste em somar dois valores e armazenar o resultado num terceiro elemento. Dado que todas as *threads* são executadas simultaneamente (ou seja, dado que é impossível dizer exactamente quando uma *thread* vai ser executada), coloca-se a questão sobre em que elemento dos vectores de entrada deve actuar cada *thread*?

O objectivo dos elementos *block* e *grid* em CUDA é o de separar os dados de entrada da função em secções lógicas e criar variáveis que as *threads* (ou neste caso a nossa *kernel*), podem ler para fazer decisões sobre que valores operar. Estas duas estruturas, denominadas `threadIdx` e `blockIdx` suportam matrizes até três dimensões, embora no caso unidimensional deste exemplo apenas se utilize a dimensão "x" da estrutura (os restantes seguem a lógica cartesiana, ou seja, são denominados por "y" e "z"). Para Compute Capability 1.3, cada *block* tem um limite de 512 *threads*, independentemente da sua organização em termos matriciais, e por sua vez uma *grid* pode conter até 65535 *blocks*, de novo independentemente da organização.

No código de exemplo foi definida uma dimensão do *block*, ou seja, que cada *block* contém `_TWIDTH` threads, e a dimensão da grid foi calculada por forma a que exista um número suficiente de *threads* para processar todos os valores dos vectores. A indexação de cada *block* repete-se, ou seja, um dado block fornece a cada thread no seu interior um valor de `threadIdx.x` entre 0 e `_TWIDTH`. Simultaneamente `blockIdx.x` é único para cada elemento *block* e logo idêntico para todo o conjunto de *threads* a que pretence. Utilizando este dois valores é possível calcular um deslocamento ou *offset* único que identifica uma

posição nos vectores em que desejamos efectuar a operação. A Figura 3.7 demonstra a ordenação de ambos os valores para o caso em que há 4 *threads* por *block* (_TWIDTH = 4). Pode-se verificar que os índices dos elementos individuais do vector podem ser calculados por $4 * blockIdx.x + threadIdx.x$.

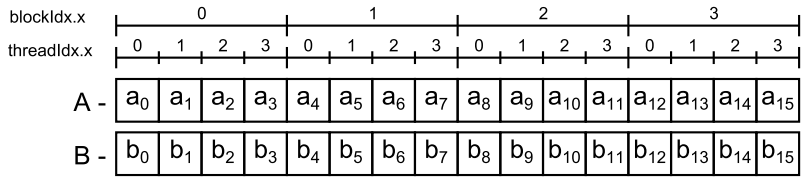

Figura 3.7: Indexação de threads e blocks

Um factor a ter em conta é a necessidade de suportar valores arbitrários de tamanho da matriz neste exemplo. Dado que uma *grid* consiste num conjunto de *blocks* de dimensão fixa o lançamento de *kernels* CUDA só permite indexar directamente múltiplos das dimensões do *block*. Para ultrapassar este problema, foi adicionada um *block* adicional à *grid*, e lógica interna na *kernel* para evitar o cálculo de valores que ultrapassem a dimensão dos vectores a calcular. Outros factores como as já mencionadas limitações de *warp* e problemas de sincronismo tornam o processo de criar uma *kernel* de CUDA com desempenho aceitável um desafio complexo. Uma introdução detalhada ao tema da criação e dimensionamento de código CUDA pode ser encontrada em [8].

3.4.5 Biblioteca CUBLAS

Incluídas com o CUDA SDK são oferecidas bibliotecas de computação matemática, destinadas a substituir a necessidade de desenvolver *kernels* para efectuar operações comuns. Uma delas é uma versão optimizada da biblioteca *Basic Linear Algebra Subprogram* (BLAS)[7] denominada *Compute Unified BLAS* ou CUBLAS e que fornece elementos básicos de cálculo vectorial e matricial. A utilização desta biblioteca permite utilizar as potencialidades do cálculo massivamente paralelo sem considerar problemas de optimização ao nível do *hardware*, assim como facilitar a conversão de algoritmos escritos originalmente

para outra plataforma. O exemplo seguinte demonstra uma implementação do algoritmo de soma de dois vectores utilizando CUBLAS:.

```
#include <cuda.h>
#include <cublas.h>

void vecSumCUBLAS(float* a, float* b, float* c, int size)
{
    int block;
    float* cublas_a;
    float* cublas_b;

    cublasInit(); // Single call required to init CUBLAS library.
    // The following statements allocate and copy memory
    // They are helper functions for cudaMalloc and cudaMemcpy
    // and as such the data can be acessed by non-CUBLAS functions
    cublasAlloc(cublas_a, size, sizeof(float));
    cublasAlloc(cublas_b, size, sizeof(float));
    cublasSetVector(size, sizeof(float), a, 1, cublas_a);
    cublasSetVector(size, sizeof(float), b, 1, cublas_b);
    cublasSaxpy(size, 1, cublas_a, 1, cublas_b, 1);    // B=A*1+B
    // The result is copied back to the CPU
    cublasGetVector(size, sizeof(float), cublas_b, c);
    //releasing the allocated data on GPU.
    cublasFree(cublas_a);
    cublasFree(cublas_b);
    cublasFree(cublas_c);
    cublasShutdown();
}
```

Apesar de ter um conjunto de funções próprias para alocação e cópia de memória estas são simplesmente macros para as funções equivalentes da API CUDA, sendo possível utilizar os resultados obtidos em operações de CUBLAS em *kernels* CUDA e vice-versa.

Capítulo 4

OMP em CUDA

Um dos factores a ter em conta quando se deseja efectuar *Compressed Sampling* é a necessidade de recuperar o sinal original após a sua aquisição, uma tarefa computacionalmente exigente. O trabalho desenvolvido presta-se a determinar as vantagens da utilização de uma plataforma CUDA para obter este resultado. Com vista a este objectivo, dois algoritmos foram enunciados, e após uma análise inicial em MATLAB, um deles foi escolhido para uma implementação em C e CUDA e análise do seu desempenho.

O enunciado do problema a que desejamos resolver é o seguinte – seja um vector $\mathbf{s}$ de dimensão W e esparsidade K, ou seja, que contenha $K << W$ elementos não-nulos. Seja uma colecção de $R << W$ vectores normalizados $\varphi = \{\varphi_1, \ldots, \varphi_r\}$ com dimensão W que denominamos por dicionário, e uma matriz Φ de dimensão $R \times W$ cujas linhas são os vectores deste dicionário. Pode definir-se como o sinal de amostragem (ou *sensing*) o sinal $\mathbf{y}$ de dimensão R, dado pelo produto da matriz Φ por $\mathbf{s}$ ou seja, $\mathbf{y} = \Phi\mathbf{s}$. Uma aplicação específica deste problema é a saida do desmodulador aleatório exemplificado na Secção 2.3.

O nosso objectivo é recuperar o sinal $\mathbf{s}$, a partir do sinal amostrado $\mathbf{y}$ e a matriz de amostragem Φ, ou seja, reconstruir o sinal K-esparso $\mathbf{s}$, através da resolução do sistema indeterminado $\mathbf{y} = \Phi\mathbf{s}$. Para resolver este problema, é necessário resolver um problema de minimização da norma l_0. Como este problema de minimização é de complexidade exponencial, uma maneira de obter uma resolução é utilizando os *greedy algorithms*, onde

podemos incluir o MP e o OMP.

4.1 Matching Pursuit

O algoritmo de Matching Pursuit tal como descrito por Mallat e Zang[10], permite encontrar iterativamente sucessivas aproximações do sinal **s**, tendo a aproximação final uma dada precisão, definida à priori:

1. **(Inicialização)**: Seja Φ o dicionário de $R \times W$, e vamos denominar por $\varphi_{1,\ldots,W}$ as colunas deste. Seja **y** o vector de entrada de dimensão r. Seja **r** um resíduo da operação tal que na primeira iteração $\mathbf{r} = \mathbf{y}$, e s' um vector nulo que vai com elementos correspondentes à aproximação do sinal de saída com dimensão W;

2. Calcular o módulo do produto interno do resíduo R_y por todos os vectores do dicionário Φ:
   $|\langle r_y, \varphi_j \rangle|$,em que $j = 1, \ldots, w$

3. Determinar o indíce λ_k para o valor máximo do produto interno entre o vector **r** e a matriz Φ : $\lambda_k = arg\ max_{j=1,\ldots W} |\langle r_y, \varphi_j \rangle|$

4. Adicionar o valor do produto interno da coluna do índice calculado com o sinal s' ao sinal de saída:
   s' = **s'** + $\langle r, \varphi_{\lambda_k} \rangle$

5. Subtrair o valor proporcional de λ_k ao resíduo Rx:
   $\mathbf{r} = \mathbf{r} - \langle \mathbf{r}, \varphi_{\lambda_k} \rangle .y_k$

6. Se a norma de **r** ainda não tiver atingido um valor pré-estabelecido de convergência, voltar para 2.

7. Uma aproximação do sinal original encontra-se em **s'**.

Uma implementação em MATLAB está disponível no Anexo A.2.

Embora este algoritmo seja computacionalmente simples, na prática são necessários um número elevado de iterações para devolver um sinal com uma correlação desejada. É então conveniente estudarmos uma modificação do algoritmo original denominado por Ortogonal Matching Pursuit.

4.2 Orthogonal Matching Pursuit

Nesta alteração, proposta por Pati[11], cada coluna φ é calculada ortogonalmente ao valor assumido de x_r, ou seja, cada índice λ_k passa a corresponder a um valor pesado de um dos vectores esparsos através da projecção do vector de entrada.

1. **(Inicialização)**: Seja Φ a matriz de $R \times W$, e vamos denominar por $\varphi_{1,\ldots,n}$ as colunas deste. Seja $\mathbf{y}$ o vector de entrada de dimensão R. Seja $\mathbf{r}$ um resíduo da operação tal que na primeira iteração $\mathbf{r} = \mathbf{y}$, e s' um vector nulo de dimensão W. Λ é um subconjunto de indíces $\Lambda \in \{1, \ldots, w\}$, inicialmente vazio e Φ uma matriz vazia que irá conter uma série de colunas φ.

2. Calcular o módulo do produto interno do resíduo $\mathbf{r}$ por todos os vectores do dicionário Φ:
   $|\langle \mathbf{r}, \varphi_j \rangle|$,em que $j = 1, \ldots, w$

3. Determinar o index λ_k do dicionário Φ para o valor máximo do produto interno:
   $\lambda_k = arg\ max_{j=1,\ldots n} |\langle \mathbf{r}, \varphi_j \rangle|$

4. Adicionar o valor de λ_k a Λ e a coluna φ referente ao índice k a Φ:
   $\Lambda = \Lambda \cup \{\lambda_k\}$, $\Psi = \begin{bmatrix} \Psi & \varphi_{\lambda_k} \end{bmatrix}$.

5. Resolver o problema de mínimos quadrados para uma nova estimativa do sinal **s'**:
   s' $= argmin_{s'} \|\mathbf{y} - \Psi_k\|_2$

6. Calcular o novo resíduo:
   $\mathbf{r} = y - \Psi_k.\mathbf{r}$

7. Se a norma de **r** ainda não tiver atingido um valor pré-estabelecido de convergência, voltar para 2.

8. A estimativa **s'** para o sinal recuperado é obtido trocando os valores presentes em **s'** pelos índices definidos em Λ.

Uma implementação em MATLAB está disponível no Anexo A.3.

4.2.1 Escolha do algoritmo OMP

As duas implementações aqui referidas em MATLAB foram utilizadas para uma análise inicial do desempenho do algoritmo, utilizando um programa que gera vectores aleatórios $\mathbf{x}$ de dimensão W com esparsidade K, e uma matriz aleatória de dimensão $R \times W$ Ψ, utilizadas como dicionário para vectores de amostragem $\mathbf{y}$. Em seguida foram realizada 100 medidas para cada valor de esparsidade, onde o vector $\mathbf{x}$ é recuperado a partir do vector $\mathbf{y}$ e da matriz Φ, utilizando quer o algoritmo de Matching Pursuit (MP) e Orthogonal Matching Pursuit (OMP). O tempo necessário para essa operação foi registado e calculada uma média para o conjunto de valores. Adicionalmente foi verificado se o algoritmo recupera o vector $\mathbf{x}$ com sucesso, sendo calculada a taxa de sucesso. Cuidado especial foi tomado para evitar resultados falsos na medição do Matching Pursuit, ou seja, para o nível de convergência considerado ($|r|^2 < 0.0001$) o número de ciclos necessários para a convergência foi menor que o limite máximo de ciclos do MP (limitado a 3000).

Os resultados obtidos demostraram que o aumento do valor de esparsidade do sinal de entrada aumenta de maneira exponencial o tempo de resolução do algoritmo MP. Na Tabela 4.1 é possível ver que isto é devido ao número de iterações necessárias para resolver o sistema na convergência desejada. O tempo de resolução do problema para ambos os algoritmos foi igualmente representado na Figura 4.1. Estes resultados sugerem que o OMP seja um algoritmo com maior potencial de optimização, dado a velocidade de resolução do problema é superior e é menos afectada pelo aumento do número de valores esparsos.

Esparsidade	Média de Iterações de MP	Tempo com MP (s)	Tempo com OMP (s)
1	2.0	0.00067	0.00155
2	51.4	0.01225	0.00076
3	81.2	0.01873	0.00089
4	528.0	0.63654	0.00137
5	620.9	0.71020	0.00156
6	933.7	1.15069	0.00207
7	2052.3	3.45534	0.00304
8	2444.8	4.89316	0.00369

Tabela 4.1: Tabela de desempenho MP e OMP para um sinal k-esparso de dimensão W=300, e dicionário R=100 linhas, calculado a partir da média de 100 realizações por valor de esparsidade

4.3 Detalhes da Implementação

Seguindo o raciocínio sugerido em [14] foi então concebida uma versão do algoritmo OMP que utiliza o método de Gram-Schmidt modificado para obter uma decomposição QR da matriz Ψ tal que $\Psi = \mathbf{Q}.\mathbf{R}$, em que $\mathbf{Q}$ é uma matriz ortogonal e $\mathbf{R}$ uma matriz triangular[12]. Esta modificação torna possível utilizar a característica das matrizes ortogonais $\mathbf{Q}^T.\mathbf{Q} = 1$ para criar uma equação que é trivial de resolver:

$\mathbf{Q}.\mathbf{R}.\mathbf{y} = \mathbf{r} \Rightarrow \mathbf{Q}^T.\mathbf{Q}.\mathbf{R}.y = \mathbf{Q}^T.\mathbf{r} \Rightarrow \mathbf{R}.\mathbf{y} = \mathbf{Q}^T.\mathbf{r}$

Dado que $\mathbf{R}$ se trata de uma matriz triangular, a equação final pode ser resolvida por um método de substituição presente nas bibliotecas BLAS. Durante a implementação deste algoritmo uma optimização da decomposição descrita em [13] foi tomada em conta, tornando apenas necessário a actualização de um único vector coluna em vez da recreação de ambas as matrizes $\mathbf{Q}$ e $\mathbf{R}$ em cada iteração. Uma implementação do algoritmo em MATLAB encontra-se disponível no Anexo A.4.

A implementação em C, baseada na implementação CBLAS da GNU Scientific Library,

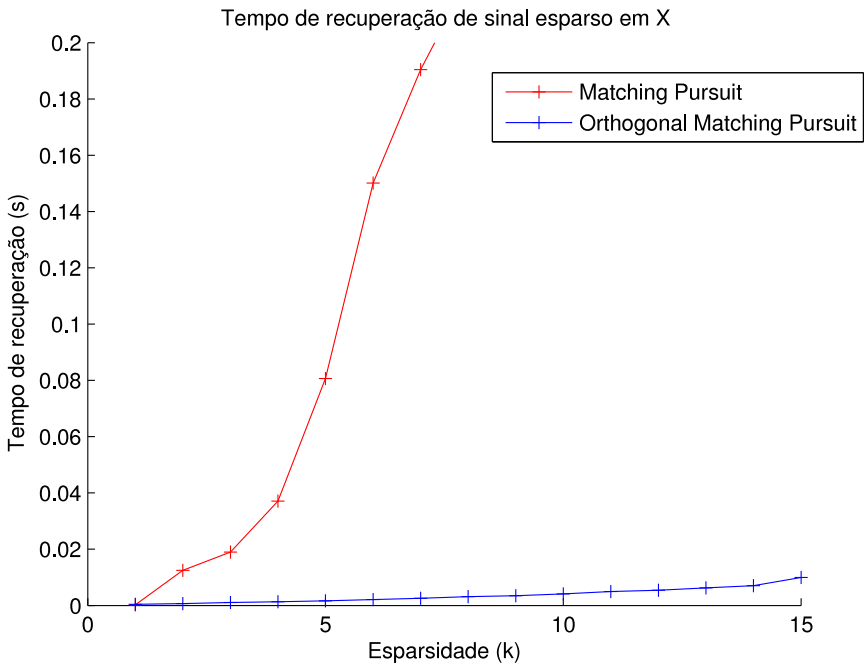

Figura 4.1: Tempo de recuperação de sinal utilizando OMP e MP, em razão do sinal k-esparso de dimensão $W = 300$, para um dicionário com $R = 100$ linhas

segue os traços gerais da implementação em MATLAB, excepto nos casos em que não existe uma implementação directa dos métodos apresentados. Um fluxograma do seu funcionamento está presente na Figura 4.2 em anexo, e o código fonte é apresentado no Anexo A.5.

A função em C foi em seguida convertida para a biblioteca CUBLAS, como apresentada no Anexo A.6. No entanto, no decorrer do processo de teste, a desempenho para valores k-esparsos elevados revelou-se abaixo do esperado, e em certos casos inferior à implementação de referência, como é possivel ver na Figura 4.3. Depois de uma análise através da ferramenta de debug Compute Profiler, foi detectado um ciclo que poderia ser optimizado:

```
cudaMemcpy(v,&d[x_size*maxDpIdx[t]], sizeof(float)*x_size,
           cudaMemcpyDeviceToDevice);
// (...)
```

```
if(t > 0)
{
  cublasSgemv('T', x_size, t, 1, Q, x_size, &mesMatrix[x_size*t],
  1, 0, &R[x_size*t], 1);
  for(int l = 0; l < t; l++)
  {
    cublasGetVector(1, sizeof(float), &R[l+x_size*t], 1, &rCache, 1);
    cublasSaxpy(x_size, -rCache, &Q[x_size*l], 1, v, 1);
  }
}
```

A função `cublasGetVector` recupera um valor da diagonal de R presente na memória do *device* e posteriormente copia-a para a variável `rCache` presente no *host*. A função `cublasSaxpy` multiplica o valor de `-rCache` por uma coluna de Q e vai consecutivamente somando esse valor a v. O valor de t aumenta um por interação do algoritmo, o que significa que se o ciclo efectuar um atraso fixo t_{delay} e a função OMP executar K ciclos para um vector **s** K-esparso, O atraso total será $t_{totaldelay} = t_{delay}.K(K+1)/2$, ou seja, para valores elevados de K torna-se significativo. Embora a implementação em CPU seja idêntica, o compilador pode efectuar operações que optimizem o ciclos a nível das instruções x86, o que não acontece no código CUDA. A solução passou pela implementação de um *kernel* para substituir o ciclo. A função efectua o mesmo processamento através de uma organização paralela:

```
#define  _TWIDTH 16
#define  P2M(x, y, max_x)      x+(y*max_x)
#define  _TX threadIdx.x
#define  _TY threadIdx.y
#define  _BX blockIdx.x*_TWIDTH
#define  _BY blockIdx.y*_TWIDTH
```

```
__global__ void
runDiagP( float* diag, float* matrix, float* output, int x_size, int y_size)
{
    __shared__ float diagS[_TWIDTH];
    __shared__ float matrixS[_TWIDTH][_TWIDTH];

    float result = output[_TX+_BX];

    for (int i = 0; i < y_size/_TWIDTH; i++)
    {
    diagS[_TY] = diag[y_size * x_size + (_TY + i*_TWIDTH)];
    matrixS[_TY][_TX] = matrix[P2M((_TX+_BX), (_TY + i*_TWIDTH), x_size)];
    __syncthreads();
    if(_TY == 0) {
        #pragma unroll
        for (int j = 0; j < _TWIDTH; j++)
        {
          result += -diagS[j] * matrixS[j][_TX];
        }
    }
    __syncthreads();
    }
    if (_TY == 0) output[_TX+_BX] = result;
}
```

Vários elementos de optimização estão presentes no *kernel*. As variáveis `diagS` e `matrixS` são criadas na memória do SM através da directiva `__shared__`, para onde as *threads* copiam elementos da diagonal de $\mathbf{R}$ e da matriz $\mathbf{Q}$ cooperativamente. Esta operação permite que vários elementos do SM acedam à memória simultâneamente. De seguida um

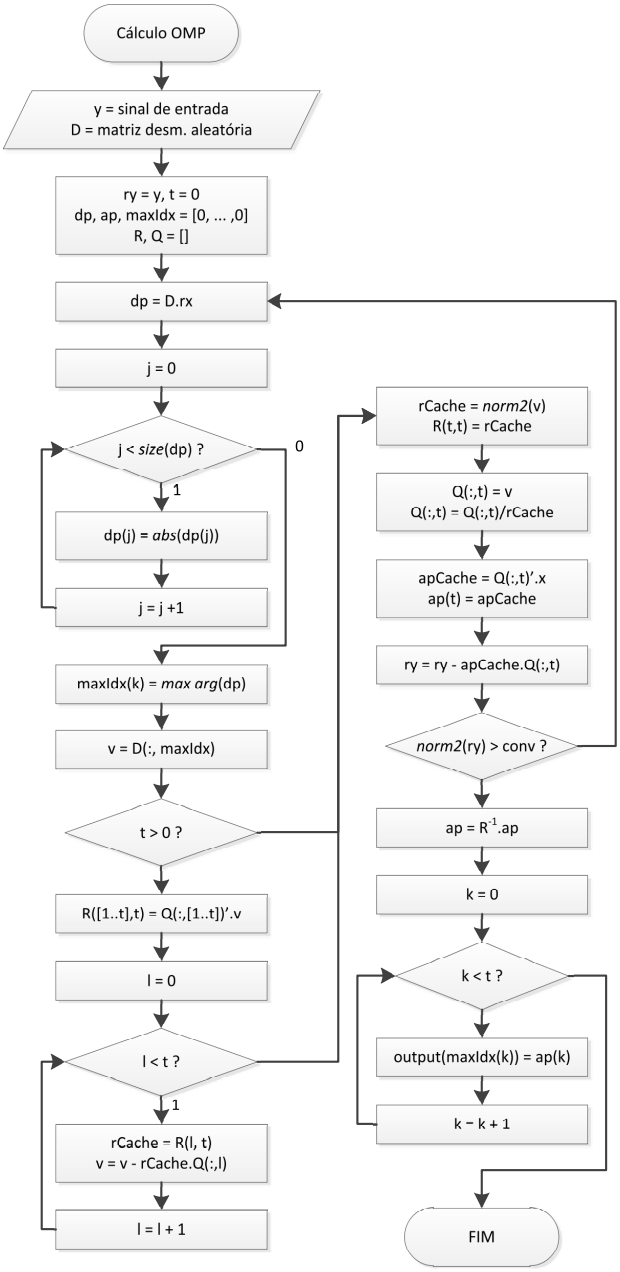

Figura 4.2: Fluxograma da função de geração de Orthogonal Matching Pursuit em C

elemento de cada *thread* é escolhido para efectuar o cálculo matemático e o comando de pre-compilador `#pragma unroll` é utilizado para sugerir ao compilador que "desenrole" o ciclo numa série de instruções individuais. A função revista no Anexo A.7 foi reescrita para executar a nova *kernel*, e os resultados obtidos são visíveis na Figura 4.3.

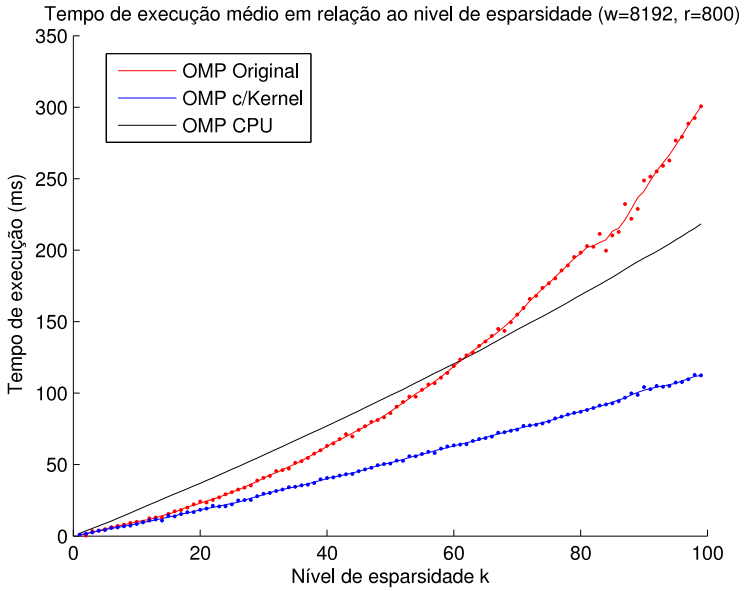

Figura 4.3: Tempo de recuperação de sinal para várias implementações de OMP em GPU

4.4 Resultados e Medições

A análise inicial dos algoritmos de OMP em GPU e CPU foi com vista a analisar o número mínimo de amostras necessárias para reconstruir um sinal K esparso. Adicionalmente este teste foi utilizado para verificar a resposta do algoritmo em GPU e CPU (ou seja, que eram equivalentes em termos de resposta). Em cada passo, foi fixado o tamanho da amostra ($W = 256$) e a esparsidade K do sinal, tendo o código sido executado 1000 vezes por passo. A taxa de sucesso em cada passo do processo foi então registada nas Figuras 4.4 e 4.5, onde podemos observar que o algoritmo apresenta resposta idêntica em ambas as soluções.

Dado que um dos objectivos desta implementação é comparar o desempenho da implementação em CUDA com uma implementação padrão em CPU, o tempo de recuperação de sinal foi objecto de um estudo mais aprofundado. Na Figura 4.6, para valores na ordem das centenas de entradas, o desempenho do GPU é geralmente inferior à do CPU. Tomando isso em conta, um pequeno teste foi efectuado para verificar a partir de que dimensão do sinal de entrada se torna viável a implementação em CUDA. Note-se que,

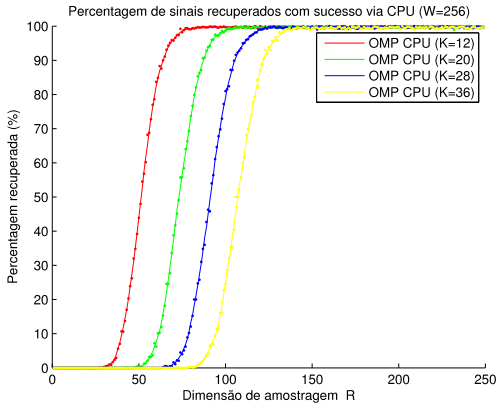

Figura 4.4: Percentagem de sinais de entrada recuperados correctamente pelo CPU, em razão de k entradas aleatórias do dicionário, para diferentes níveis de esparsidade do sinal de entrada.

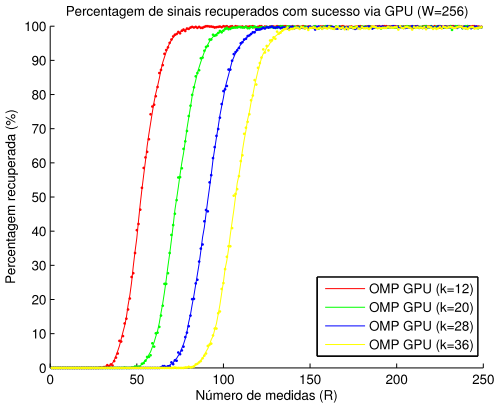

Figura 4.5: Percentagem de 1000 sinais de entrada recuperados correctamente pelo GPU, em razão de k entradas aleatórias do dicionário, para diferentes níveis de esparsidade do sinal de entrada.

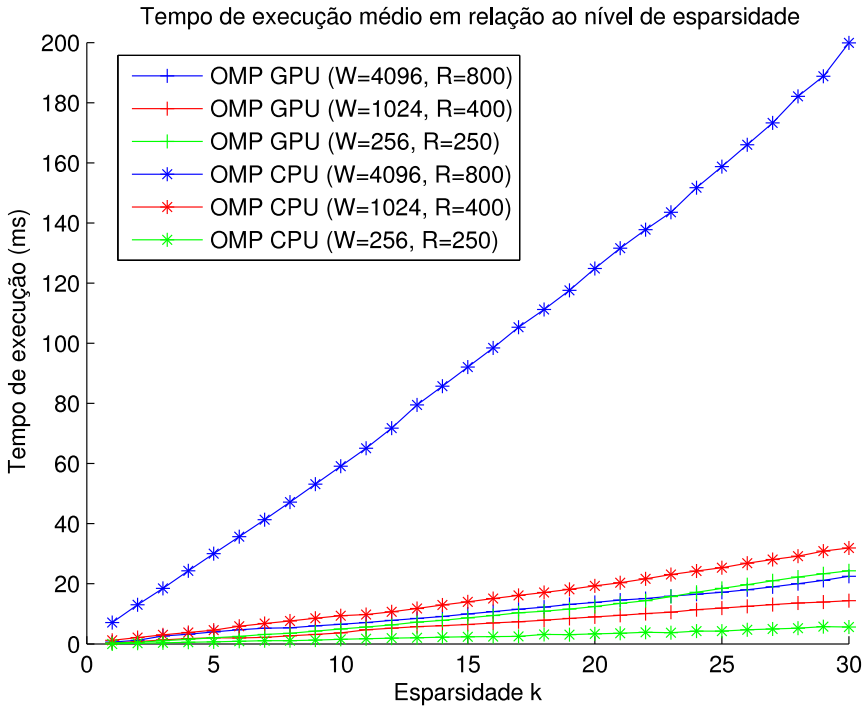

Figura 4.6: Tempo médio de resolução do OMP, para diferentes valores de esparsidade do sinal de entrada (calculado pela média de 1000 valores por passo).

ao aumentar a dimensão W do sinal mantendo o valor de esparsidade K fixo estamos directamente a afectar a dimensão da matriz Φ efectuada nos cálculos, não o número de iterações efectuadas pelo OMP. A Tabela 4.2 mostra que para valores relativamente baixos de amostragem e esparsidade, os cálculos em GPU tornam-se tão eficazes como o CPU a partir de aproximadamente 4096 entradas.

Dimensão(w)	Exec. CPU(ms)	Exec. GPU(ms)	Razão CPU/GPU
1024	15.25	53.06	0.287
2048	30.88	56.36	0.548
3072	43.04	58.72	0.734
4096	60.25	60.09	1.003
5120	75.51	62.97	1.199
6144	93.34	65.15	1.433
7168	108.51	65.95	1.645
8128	122.11	67.91	1.798

Tabela 4.2: Tempo médio de processamento de um sinal em OMP com amostragem R=200 e esparsidade K=30, (calculado a partir de 100 medições)

Esta redução de tempo é igualmente visível na Figura 4.7, que correlaciona o tempo de execução do CPU e do GPU. Podemos verificar que para valores superiores, o GPU apresenta um ganho considerável em relação à implementação original, mesmo ao aumentar o número de amostras R.

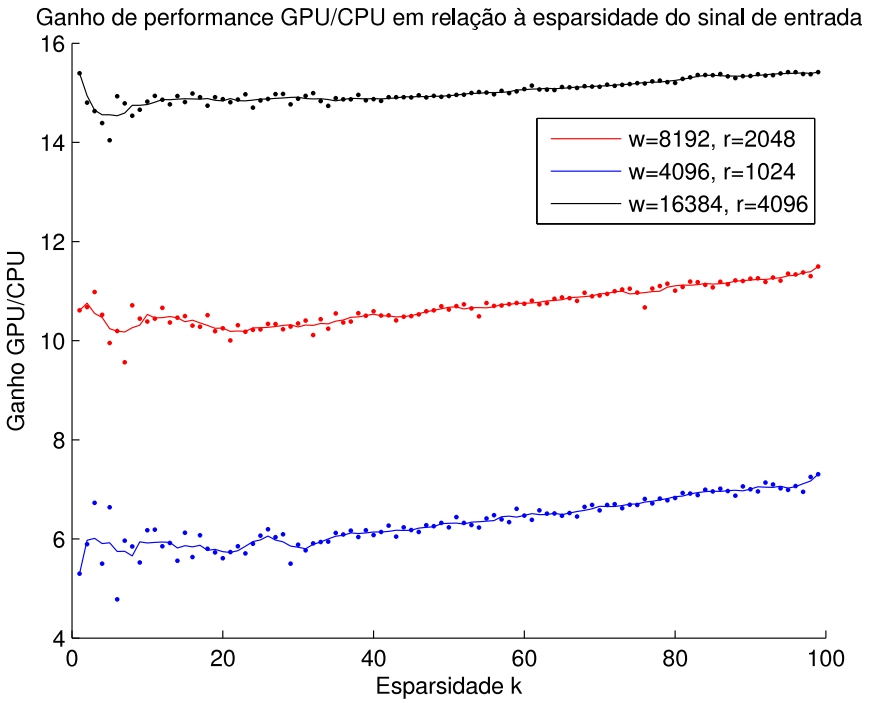

Figura 4.7: Relação entre tempo de execução para os algoritmos OMP em CPU e GPU, calculado a partir de 100 medições por ponto e medido para diferentes valores de esparsidade do sinal de entrada

Capítulo 5

Conclusões

Os resultados obtidos apontam para uma boa prestação do algoritmo OMP quando programado em CUDA. É possível observar uma melhoria significativa no tempo de resolução da problema esparso, e existem ainda vários campos em que se pode expandir o estudo de utilização de GPUs. Por exemplo, apenas um GPU foi utilizado para realizar a operação de OMP, mas o estudo de um método de OMP (ou outro "greedy algorythm") que utilize os quatro processadores presentes na plataforma Tesla pode ser um caminho para obter melhores resultados.

A implementação de mais *kernels* que reduzam a transferência de dados entre *host* e *device*, como o efectuado neste documento, pode igualmente levar a um aumento da velocidade de resolução do problema. É ainda de considerar a possibilidade de utilizar a tecnologia *stream* de CUDA para faciliar o envio de informação em tempo real de e para o dispositivo, com o objectivo de testar a capacidade e latência do algoritmo quando utilizado como um desmodulador, por exemplo.

Apêndice A

Código Fonte

A.1 randomdemod.m

```
% Creation of a frequency-sparse signal into a time domain, generation
% of demodulation matrixes, and random demodulation technique
% Nuno Antunes, 2011 UA
%
% References:
% José Vieira. Random Demodulation, 2011 UA
% Joel A.Tropp, Jason N. Laska, Marco F. Duarte, Justin K. Romberg and
%   Richard G. Braniuk. Beyond Nyquist: Efficient sampling of sparse
%   bandlimited signals. IEEE Transactions on Information Theory,
%   56(1):520-544, 210

clear all; close all;

% W - Valor da frequência mais elevada que desejamos amostrar
W = 2000;
% R - Valor da amostragem efectuada
```

```
R = 400;
% tn - número de amostras efectuadas
tn = W/R

% geração de um sinal de esparsidade fixa na frequência.
signal_gen = zeros(W,1);
sparsity = 50; % number of non-zero values
indexes = ceil(rand(sparsity, 1)*ceil(size(signal_gen, 1)/2))
values = 1/sparsity;
signal_gen(indexes) = values;
x = real(ifft(signal_gen))*W;

plot(x(1:100), 'r*-');
hold on;

% geração das matrizes de desmodulação
pt = (rand(W, 1)*2) - 1; % valor aleatório que é multiplicado pelo
                         % sinal de entrada
n = [0:W-1];
omg = [-(W/2)+1:(W/2)-1];
omg = [omg W/2];
F = (1/sqrt(W))*(exp(-j*2*pi*n'*omg/W));
Fr = real(F);   % versão "real" da matriz de Fourier permutada
P = diag(pt);
idx_i = 1;
for I = 1:R
    idx_f = idx_i + (ceil(W/R) - 1);
    H(I, idx_i:idx_f) = 1;
    idx_i = idx_f + 1;
```

```
end

M = H*P;    % matriz de aquisição do sinal
sig = M*Fr;  % matriz de desmodelação aleatória

%aquisição do sinal à razão R

% sampling
y = M*x

% uma versão "iterativa" da amostragem do sinal.
% for I = 1:tn:W
%     block = signal(I:I+tn-1).*pt(I:I+tn-1);
%     y = [x; mean(block)];
% end

% visualização do valor do sinal amostrado no tempo
for I = 1:size(y,1)
    samp(((I-1)*tn)+1:I*tn) = y(I);
end
plot(samp(1:100), 'g');

% recuperação do sinal - note-se que só o sinal y está a ser utilizado
% e que sig não contem informação do sinal de entrada
[out ~] = omp_basic(y, sig, 0);

xr = Fr*out; %reconstrução do sinal pela matriz de coeficientes

plot(real(xr(1:100)), 'b');
```

```
stem(pt(1:100), 'co');

title('Sinal no domínio do tempo (detalhe)');
xlabel('Tempo (ms)');
ylabel('Amplitude');
legend('Sinal de Entrada', 'Sinal Amostrado', 'Sinal Reconstruido',
       'Valor aleatório');
hold off;

figure;
hold on;
plot(x, 'r*-');
plot(samp, 'g');
plot(real(xr), 'b');
title('Sinal no domínio do tempo');
xlabel('Tempo (ms)');
ylabel('Amplitude');
legend('Sinal de Entrada', 'Sinal Amostrado', 'Sinal Reconstruido');
hold off;

figure;
axis([-W/2 W/2 0 1]);
freq = [-W/2:W/2-1];
sad = fftshift(x);
stem(freq, fftshift(fft(x)), 'r*');
hold on;
stem(freq, fftshift(fft(xr)), 'b');
title('Sinal no domínio da frequência');
xlabel('Frequência (Hz)');
```

```
ylabel('Amplitude');
legend('Sinal de Entrada', 'Sinal Reconstruido');
hold off;
```

A.2 mpbasic.m

```
function [dpk Rx] = mp_basic(x, D, k)
% mp_basic : performs matching pursuit algorithm (reference)

% USAGE:
% mp_basic(x, D, k)
% PARAMETERS:
% x : entry data vector
% D : dictionary (normalized!)
% k : maximum number of intertions to perform
% OUTPUT:
% Rf : vector of the residual
% fk : actual vector aproximation
% dpk : solution

% Reference:
% Pati, Y. C., R. Rezaiifar, et al. (1993). "Orthogonal Matching
%     Pursuit - Recursive Function Approximation with Applications to
%     Wavelet Decomposition." Conference Record of the Twenty-Seventh
%     Asilomar Conference on Signals, Systems & Computers, Vols 1
%     and 2: 40-44 1659.

n = size(D);
```

```
 Rx(:, 1) = x; % residual(error)
 t_dpk = 0;
 dpk = zeros(n(2), 1);
 dp = zeros(n(2), 1);

I = 1;
% This function runs k interactions.
while (I < k & norm(Rx(:, I)) > 0.000001)
    % Compute the inner-products for all dictionary entries
    J=1:n(2);
    dp(J)=(Rx(:,I)'*D(:, J));
    % Find nk+1 so that dp(nk+1) >= max(dp)
    [~, nk] = max(abs(dp)); % index for the best correlation
    t_dpk = dp(nk(1)); % inner-product from the index
    Rx(:, I+1) = Rx(:, I) - t_dpk*D(:, nk(1));  % residue
    dpk(nk(1)) = dpk(nk(1)) + t_dpk;
I = I + 1;
end
```

A.3 ompbasic.m

```
function [dpk Rx] = omp_basic(x, D, conv)
% omp_basic : performs ortogonal matching pursuit algorithm (reference)
% USAGE:
% omp_basic(f, conv)
% PARAMETERS:
% x : entry data vector
% D : dictionary
% conv : convergence value
```

```
% OUTPUT:
% Rx : vector of the residual
% dpk : solution

% References:
% Tropp, J. A. and A. C. Gilbert (2007). "Signal recovery from random
%     measurements via orthogonal matching pursuit." Ieee Transactions
%     on Information Theory 53(12): 4655-4666.

 n = size(D);

 Rx(:, 1) = x; % residual(error)
% t_dpk = zeros(n(2), 1);
 dpk = zeros(n(2), 1);
 index_set = [];
 measurement_matrix = [];
 t = 1;
 dp = zeros(n(2), 1);
% This function runs at maximum t interactions.
while (t < 10000 & norm(Rx(:, t)) > 0.000001)
    % Compute the inner-products for all dictionary entries
    J=1:n(2);
    dp(J)=(Rx(:,t)'*D(:, J));
    % Find nk+1 so that dp(nk+1) >= opti*max(dp)
    [~, nk] = max(abs(dp));
    index_set = [index_set nk(1)];
    measurement_matrix = [measurement_matrix D(:,nk(1))];
    Dt = measurement_matrix\x;   %solves the least squares problem
    ap = measurement_matrix*Dt;
```

```
    t = t + 1;  %Augment iteration
    Rx(:, t) = x - ap;
end
dpk(index_set) = Dt;
}
```

A.4 ompqr.m

```
function [dpk Rx] = omp_qr(x, D, conv)
% omp_basic : performs ortogonal matching pursuit algorithm (QR matrix)
% USAGE:
% omp_qr(f, conv)
% PARAMETERS:
% x : entry data vector
% D : dictionary
% conv : convergence value
% OUTPUT:
% Rx : vector of the residual
% dpk : solution
% References:
% Tropp, J. A. and A. C. Gilbert (2007). "Signal recovery from random
%     measurements via orthogonal matching pursuit." Ieee Transactions
%     on Information Theory 53(12): 4655-4666.

 n = size(D);
 Rx(:, 1) = x; % residual(error)
 dpk = zeros(n(2), 1);
 index_set = [];
 measurement_matrix = [];
```

```
 t = 1;
 dp = zeros(n(2), 1);
 ap2 = zeros(n(2), 1);
 bl = x;
 q1 = zeros(n(1),1);
 r1 = zeros(n(1),1);
 R=[];
 Q=[];
 % This function runs at maximum t interactions.
while (t < 10000 & norm(Rx(:, t)) > 0.000001)
    % Compute the inner-products for all dictionary entries
    J=1:n(2);
    dp(J)=(Rx(:, t)'*D(:, J));
    [~, nk] = max(abs(dp));
    index_set = [index_set nk(1)];
    measurement_matrix = [measurement_matrix D(:,nk(1))];
    D(:,nk(1)) = zeros(n(1), 1);
    v=measurement_matrix(:,t);
    for w=1:t-1
        r1(w)=Q(:,w)'*measurement_matrix(:,t);
        v=v-r1(w)*Q(:,w);
    end
    r1(t)=norm(v);
    q1=v/r1(t);
    R = [R r1];
    Q = [Q q1];
    ap2(t) = Q(:,t)'*x;
    t = t + 1;  %Augment iteration
    Rx(:, t) = Rx(:, t-1) - ap2(t-1)*Q(:,t-1);
```

```
end
Dt = R(1:t-1, 1:t-1)\ap2(1:t-1);
dpk(index_set) = Dt;
```

A.5 csampling_math.cpp

```
// csampling_math.c - CPU Orthogonal Matching Pursuit
// Nuno Antunes - 2010

#include <math.h>
#include <limits.h>
#include <time.h>
#include <gsl/gsl_cblas.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_linalg.h>

#include "csampling.h"

// exernal declarations
extern "C" void doOMP_cpu(float* x, float* dictionary, float* output,
                          unsigned int x_size, unsigned int y_size);
// internal declarations
int orthogonalMP_cpu(gsl_vector_float* x,  gsl_matrix_float* d,
                     gsl_vector_float* output);

void doOMP_cpu(float* x, float* dictionary, float* output,
               unsigned int x_size, unsigned int y_size)
```

```
{
    unsigned int i, j, errchk, sprchk;
    gsl_vector_float* gsl_x = gsl_vector_float_alloc(x_size);
    gsl_matrix_float* gsl_d = gsl_matrix_float_alloc(x_size, y_size);
    gsl_vector_float* gsl_out = gsl_vector_float_alloc(y_size);
    gsl_vector_float_set_zero(gsl_out);
    for(i = 0; i < x_size; i++)
    {
       gsl_vector_float_set(gsl_x, i, x[i]);
    }
    for(i = 0; i < x_size; i++)
    {
        for(j = 0; j < y_size; j++)
        {
            gsl_matrix_float_set(gsl_d, i, j, dictionary[P2M(i, j, x_size)]);
        }
    }
    errchk = orthogonalMP_cpu(gsl_x, gsl_d, gsl_out, time);
    gsl_vector_float_free(gsl_x);
    gsl_matrix_float_free(gsl_d);
    gsl_vector_float_free(gsl_out);
}

int orthogonalMP_cpu(gsl_vector_float* x, gsl_matrix_float* d,
                     gsl_vector_float* output, mpbench* time)
{
    int result = 0;
    int t = 0;
    float dpCache, rCache, apCache;
```

```
float prevSnrm;
int* maxDpIdx;
float *testMatrix;

maxDpIdx = (int*) malloc(sizeof(int)*d->size1);

gsl_vector_float* rx = gsl_vector_float_alloc(x->size);
gsl_vector_float* dp = gsl_vector_float_alloc(d->size2);
gsl_vector_float* ap = gsl_vector_float_alloc(d->size2);
gsl_vector_float* v = gsl_vector_float_alloc(d->size1);
gsl_vector_float_set_zero(ap);

int m_size = (d->size1 < d->size2) ? d->size2 : d->size1/2;

gsl_matrix_float* mesMatrix = gsl_matrix_float_alloc(d->size1,m_size);
gsl_matrix_float* R = gsl_matrix_float_alloc(d->size1,m_size);
gsl_matrix_float* Q = gsl_matrix_float_alloc(d->size1,m_size);
gsl_vector_float_memcpy(rx, x);
gsl_matrix_float_set_zero(mesMatrix); // set rx = x
gsl_matrix_float_set_zero(R); // set matrix R to zero
ptimer = clock();
bool accurancyCheck = true;
prevSnrm = gsl_blas_snrm2(rx);
while((t < d->size1) && ((gsl_blas_snrm2(rx) > 0.0001) || t == 0)
      && accurancyCheck)
{
    result = gsl_blas_sgemv(CblasTrans, 1, d, rx, 0, dp);
    for(unsigned int j = 0; j < dp->size; j++)
        gsl_vector_float_set(dp, j, fabs(gsl_vector_float_get(dp, j)));
```

```
    maxDpIdx[t] = gsl_vector_float_max_index(dp);
    gsl_matrix_float_set_col(mesMatrix, t,
        &gsl_matrix_float_const_column(d, maxDpIdx[t]).vector);
    gsl_vector_float_memcpy(v,
        &gsl_matrix_float_const_column(d, maxDpIdx[t]).vector);
    if(t > 0)
    {
          gsl_blas_sgemv(CblasTrans, 1,
          &gsl_matrix_float_const_submatrix(Q, 0, 0, d->size1, t).matrix,
          v, 0,
          &gsl_matrix_float_subcolumn(R, t, 0, t).vector);
        for(int l = 0; l < t; l++)
        {
            rCache = gsl_matrix_float_get(R, l, t);
            gsl_blas_saxpy(-rCache,
                           &gsl_matrix_float_const_column(Q, l).vector, v);
        }
    }
    rCache = gsl_blas_snrm2(v);
        gsl_matrix_float_set(R, t, t, rCache);
    gsl_vector_float_memcpy(&gsl_matrix_float_const_column(Q,t).vector, v);
    gsl_blas_sscal(1/rCache, &gsl_matrix_float_const_column(Q,t).vector);
    gsl_blas_sdot(&gsl_matrix_float_const_column(Q,t).vector, x, &apCache);
    gsl_vector_float_set(ap, t, apCache);
    gsl_blas_saxpy(-apCache, &gsl_matrix_float_const_column(Q,t).vector, rx);
    t++;
    prevSnrm = gsl_blas_snrm2(rx);
}
gsl_blas_strsv(CblasUpper, CblasNoTrans, CblasNonUnit,
```

```
                    &gsl_matrix_float_const_submatrix(R, 0, 0, t, t).matrix,
                    &gsl_vector_float_subvector(ap, 0,t).vector);
    for(unsigned int k = 0; k < t; k++)
    {
        gsl_vector_float_set(output, maxDpIdx[k], gsl_vector_float_get(ap, k));
    }
    free(maxDpIdx);
    gsl_vector_float_free(rx);
    gsl_vector_float_free(dp);
    gsl_vector_float_free(ap);
    gsl_vector_float_free(v);
    gsl_matrix_float_free(mesMatrix);
    gsl_matrix_float_free(R);
    gsl_matrix_float_free(Q);
    return 0;
}
```

A.6 csampling_cuda.cu (sem utilizar uma kernel optimizada)

```
// csampling_cuda.cu - CUDA functions for Compressed Samping

#include <cstdlib>
#include <stdio.h>
#include <math.h>
#include <cublas.h>
#include <cuda_runtime_api.h>
```

```
#include "csampling.h"

// exernal declarations
extern "C" void doOMP_cublas(float* x, float* dictionary, float* output,
                             int x_size, int y_size);
// internal declarations
int orthogonalMP_cublas(float* x,  float* d, float* output, int x_size,
                        int y_size, mpbench* timers);

void doOMP_cublas(float* x, float* dictionary, float* output, int x_size,
                  int y_size, mpbench* timers, bool use_kernel)
{
    int i, j, errchk, sprchk;
    float* cublas_x;
    float* cublas_d;
    float* cublas_o;
    cublasInit();
    errchk = 0;
    cublasAlloc(x_size, sizeof(x[0]), (void**)&cublas_x);
    cublasAlloc(x_size*y_size, sizeof(dictionary[0]), (void**)&cublas_d);
    cublasAlloc(y_size, sizeof(output[0]), (void**)&cublas_o);
    cublasSetVector(x_size, sizeof(x[0]), x, 1, cublas_x, 1);
    cublasSetMatrix(x_size, y_size, sizeof(dictionary[0]), dictionary,
                    x_size, cublas_d, x_size);
    cudaMemset(cublas_o, 0, y_size*sizeof(float));
    cublasGetError();

    errchk = orthogonalMP_cublas(cublas_x, cublas_d, cublas_o,
                                       x_size, y_size);
```

```
    cublasGetVector(y_size, sizeof(float), cublas_o, 1, output, 1);
    cublasFree(cublas_x);
    cublasFree(cublas_d);
    cublasFree(cublas_o);
}

int orthogonalMP_cublas(float* x, float* d, float* output, int x_size,
                        int y_size)
{
    int result = 0;
    int t = 0;
    float dpCache, rCache, apCache;
    int* maxDpIdx;
    float *testMatrix;

    float *rx; // residue for the entry signal
    float *dp;
    float *ap; // signal aproximation for X
    float *mesMatrix;
    float *R, *Q;
    float *v;
    float *dpCPU;
    float *r1;
    float *vk;

    ptimer = clock();
    dpCPU = (float*) malloc(sizeof(float)*y_size);
    maxDpIdx = (int*) malloc(sizeof(int)*x_size);
    testMatrix = (float*) malloc(sizeof(float)*x_size*y_size);
```

```
cublasAlloc(x_size, sizeof(float), (void**)&rx);
cublasAlloc(y_size, sizeof(float), (void**)&dp);
cublasAlloc(x_size, sizeof(float), (void**)&ap);
cublasAlloc(x_size, sizeof(float), (void**)&v);
cudaMemset(ap, 0, sizeof(float)*x_size);    // set ap vector to 0

// This allows test cases where the signal is smaller
// than the number of dictionary rows
int m_size = (y_size < x_size) ? x_size : y_size/2;

cublasAlloc(x_size*m_size, sizeof(float), (void**)&mesMatrix);
cublasAlloc(x_size*m_size, sizeof(float), (void**)&R);
cublasAlloc(x_size*m_size, sizeof(float), (void**)&Q);

cudaMemset(R, 0, sizeof(float)*x_size*m_size); // vector R = 0
cudaMemcpy(rx, x, sizeof(float)*x_size, cudaMemcpyDeviceToDevice);
while (t < x_size && cublasSnrm2(x_size, rx, 1) > 0.0001)
{
    cublasSgemv('T', x_size, y_size, 1, d, x_size, rx, 1, 0, dp, 1);
    cublasGetVector(y_size, sizeof(float), dp, 1, dpCPU, 1);
    for(unsigned int k = 0; k < y_size; k++)
        dpCPU[k] = fabs(dpCPU[k]);
    cublasSetVector(y_size, sizeof(float), dpCPU, 1, dp, 1);
    maxDpIdx[t] = cublasIsamax(y_size, dp, 1) - 1;
    cudaMemcpy(&mesMatrix[x_size*t],&d[x_size*maxDpIdx[t]],
               sizeof(float)*x_size, cudaMemcpyDeviceToDevice);
    cudaMemcpy(v,&d[x_size*maxDpIdx[t]], sizeof(float)*x_size,
               cudaMemcpyDeviceToDevice);
```

```
    cudaMemcpy(vk,&d[x_size*maxDpIdx[t]], sizeof(float)*x_size,
               cudaMemcpyDeviceToDevice);
    if(t > 0)
    {
        cublasSgemv('T', x_size, t, 1, Q, x_size, &mesMatrix[x_size*t],
                    1, 0, &R[x_size*t], 1);
        for(int l = 0; l < t; l++)
        {
            cublasGetVector(1, sizeof(float), &R[l+x_size*t], 1,
                            &rCache, 1);
            cublasSaxpy(x_size, -rCache, &Q[x_size*l], 1, v, 1);
        }
    }
    rCache = cublasSnrm2(x_size, v, 1);
    cublasSetVector(1, sizeof(float), &rCache, 1, &R[t+x_size*t], 1);
    cudaMemcpy(&Q[x_size*t],v, sizeof(float)*x_size,
               cudaMemcpyDeviceToDevice);
    cublasSscal(x_size, 1/rCache, &Q[x_size*t], 1);
    cublasSgemv('T', x_size, 1, 1, &Q[x_size*t], x_size, x, 1, 0,
                &ap[t], 1);
    cublasGetVector(1, sizeof(float), &ap[t], 1, &apCache, 1);
    cublasSaxpy(x_size, -apCache, &Q[x_size*t], 1, rx, 1);
    t++;
}
cublasStrsv('U', 'N', 'N', t, R, x_size, ap, 1);
for(int k = 0; k < t; k++)
{
    cudaMemcpy(&output[maxDpIdx[k]], &ap[k], sizeof(float),
               cudaMemcpyDeviceToDevice);
```

```
    }
    free(dpCPU);
    free(maxDpIdx);
    free(testMatrix);
    cublasFree(rx);
    cublasFree(dp);
    cublasFree(ap);
    cublasFree(v);
    cublasFree(mesMatrix);
    cublasFree(R);
    cublasFree(Q);
    return result;
}
```

A.7 csampling_kernel.cu

```
// csampling_kernel.cpp - GPU kernel library for the Compressed Samping project
// Nuno Antunes - 2011

#ifndef _CSAMPLING_KERNEL_H_
#define _CSAMPLING_KERNEL_H_

#define  _TWIDTH 16
#define  P2M(x, y, max_x)      x+(y*max_x)
#define  _TX threadIdx.x
#define  _TY threadIdx.y
#define  _BX blockIdx.x*_TWIDTH
#define  _BY blockIdx.y*_TWIDTH
```

```
#include <stdio.h>

__global__ void
runDiagP( float* diag, float* matrix, float* output, int x_size, int y_size);
__global__ void
runDiagS( float* diag, float* matrix, float* output, int x_size, int y_size,
          int offset);

void doQRDiag(float* rw, float* Q, float* v, int x_size, int t)
{
    int block_x, block_y, rem;

    block_x = x_size / _TWIDTH;
    block_y = t / _TWIDTH;
rem = t % _TWIDTH;
    if(block_y)
    {
  dim3 dimBlock(_TWIDTH, _TWIDTH);
      dim3 dimGrid(block_x, 1);
      runDiagP <<< dimGrid, dimBlock >>> (rw, Q, v, x_size, t);
      cudaThreadSynchronize();
    }
    if (rem)
    {
      dim3 dimBlockS(_TWIDTH, rem);
      dim3 dimGridS(block_x, 1);
      runDiagS <<< dimGridS, dimBlockS >>> (rw, Q, v, x_size, rem,
                                            (block_y * _TWIDTH));
      cudaThreadSynchronize();
```

```
    }
}

__global__ void
runDiagP( float* diag, float* matrix, float* output, int x_size, int y_size)
{
    __shared__ float diagS[_TWIDTH];
    __shared__ float matrixS[_TWIDTH][_TWIDTH];

    float result = output[_TX+_BX];

    for (int i = 0; i < y_size/_TWIDTH; i++)
    {
    diagS[_TY] = diag[y_size * x_size + (_TY + i*_TWIDTH)];
    matrixS[_TY][_TX] = matrix[P2M((_TX+_BX), (_TY + i*_TWIDTH), x_size)];
    __syncthreads();
    if(_TY == 0) {
        #pragma unroll
        for (int j = 0; j < _TWIDTH; j++)
        {
          result += -diagS[j] * matrixS[j][_TX];
        }
    }
    __syncthreads();
    }
    if (_TY == 0) output[_TX+_BX] = result;
}

__global__ void
```

```
runDiagS( float* diag, float* matrix, float* output, int x_size, int y_size,
          int offset)
{
    __shared__ float diagS[_TWIDTH];
    __shared__ float resultS[_TWIDTH];
    __shared__ float matrixS[_TWIDTH][_TWIDTH];

    float  result = output[_TX+_BX];

    diagS[_TY] = diag[(y_size+offset)*x_size+(_TY+offset)];
    matrixS[_TY][_TX] =  matrix[P2M((_TX+_BX), (offset+_TY), x_size)];
    __syncthreads();
    if(_TY == 0) {
      #pragma unroll
      for (int j = 0; j < y_size; j++)
      {
        result += -diagS[j] * matrixS[j][_TX];
      }
      output[_TX+_BX] = result;
    }
    __syncthreads();
}

#endif // #ifndef _CSAMPLING_KERNEL_H_
```

A.8 csampling_cuda.cu (com kernel)

```
// csampling_cuda.cu - CUDA functions for Compressed Samping
```

```
#include <cstdlib>
#include <stdio.h>
#include <math.h>
#include <cublas.h>
#include <cuda_runtime_api.h>

#include "csampling.h"

// includes, kernels
#include <csampling_kernel.cu>

// exernal declarations
extern "C" void doOMP_cublas(float* x, float* dictionary, float* output,
                             int x_size, int y_size);
// internal declarations
int orthogonalMP_cublas(float* x,  float* d, float* output, int x_size,
                        int y_size, mpbench* timers);

void doOMP_cublas(float* x, float* dictionary, float* output, int x_size,
                  int y_size, mpbench* timers, bool use_kernel)
{
    int i, j, errchk, sprchk;
    float* cublas_x;
    float* cublas_d;
    float* cublas_o;
    cublasInit();
    errchk = 0;
    cublasAlloc(x_size, sizeof(x[0]), (void**)&cublas_x);
    cublasAlloc(x_size*y_size, sizeof(dictionary[0]), (void**)&cublas_d);
```

```
    cublasAlloc(y_size, sizeof(output[0]), (void**)&cublas_o);
    cublasSetVector(x_size, sizeof(x[0]), x, 1, cublas_x, 1);
    cublasSetMatrix(x_size, y_size, sizeof(dictionary[0]), dictionary,
                    x_size, cublas_d, x_size);
    cudaMemset(cublas_o, 0, y_size*sizeof(float));
    cublasGetError();

    errchk = orthogonalMP_cublas(cublas_x, cublas_d, cublas_o,
                                        x_size, y_size);
    cublasGetVector(y_size, sizeof(float), cublas_o, 1, output, 1);
    cublasFree(cublas_x);
    cublasFree(cublas_d);
    cublasFree(cublas_o);
}

int orthogonalMP_cublas(float* x, float* d, float* output, int x_size,
                        int y_size)
{
    int result = 0;
    int t = 0;
    float dpCache, rCache, apCache;
    int* maxDpIdx;
    float *testMatrix;

    float *rx; // residue for the entry signal
    float *dp;
    float *ap; // signal aproximation for X
    float *mesMatrix;
    float *R, *Q;
```

```
float *v;
float *dpCPU;
float *r1;
float *vk;

ptimer = clock();
dpCPU = (float*) malloc(sizeof(float)*y_size);
maxDpIdx = (int*) malloc(sizeof(int)*x_size);
testMatrix = (float*) malloc(sizeof(float)*x_size*y_size);

cublasAlloc(x_size, sizeof(float), (void**)&rx);
cublasAlloc(y_size, sizeof(float), (void**)&dp);
cublasAlloc(x_size, sizeof(float), (void**)&ap);
cublasAlloc(x_size, sizeof(float), (void**)&v);
cudaMemset(ap, 0, sizeof(float)*x_size);    // set ap vector to 0

// This allows test cases where the signal is smaller
// than the number of dictionary rows
int m_size = (y_size < x_size) ? x_size : y_size/2;

cublasAlloc(x_size*m_size, sizeof(float), (void**)&mesMatrix);
cublasAlloc(x_size*m_size, sizeof(float), (void**)&R);
cublasAlloc(x_size*m_size, sizeof(float), (void**)&Q);

cudaMemset(R, 0, sizeof(float)*x_size*m_size); // vector R = 0
cudaMemcpy(rx, x, sizeof(float)*x_size, cudaMemcpyDeviceToDevice);
while (t < x_size && cublasSnrm2(x_size, rx, 1) > 0.0001)
{
    cublasSgemv('T', x_size, y_size, 1, d, x_size, rx, 1, 0, dp, 1);
```

```
    cublasGetVector(y_size, sizeof(float), dp, 1, dpCPU, 1);
    for(unsigned int k = 0; k < y_size; k++)
        dpCPU[k] = fabs(dpCPU[k]);
    cublasSetVector(y_size, sizeof(float), dpCPU, 1, dp, 1);
    maxDpIdx[t] = cublasIsamax(y_size, dp, 1) - 1;
    cudaMemcpy(&mesMatrix[x_size*t],&d[x_size*maxDpIdx[t]],
               sizeof(float)*x_size, cudaMemcpyDeviceToDevice);
    cudaMemcpy(v,&d[x_size*maxDpIdx[t]], sizeof(float)*x_size,
               cudaMemcpyDeviceToDevice);
    cudaMemcpy(vk,&d[x_size*maxDpIdx[t]], sizeof(float)*x_size,
               cudaMemcpyDeviceToDevice);
    if(t > 0)
    {
        cublasSgemv('T', x_size, t, 1, Q, x_size, &mesMatrix[x_size*t],
                    1, 0, &R[x_size*t], 1);
        doQRDiag(R, Q, v, x_size, t);
    }
    rCache = cublasSnrm2(x_size, v, 1);
    cublasSetVector(1, sizeof(float), &rCache, 1, &R[t+x_size*t], 1);
    cudaMemcpy(&Q[x_size*t],v, sizeof(float)*x_size,
               cudaMemcpyDeviceToDevice);
    cublasSscal(x_size, 1/rCache, &Q[x_size*t], 1);
    cublasSgemv('T', x_size, 1, 1, &Q[x_size*t], x_size, x, 1, 0,
                &ap[t], 1);
    cublasGetVector(1, sizeof(float), &ap[t], 1, &apCache, 1);
    cublasSaxpy(x_size, -apCache, &Q[x_size*t], 1, rx, 1);
    t++;
}
cublasStrsv('U', 'N', 'N', t, R, x_size, ap, 1);
```

```
    for(int k = 0; k < t; k++)
    {
        cudaMemcpy(&output[maxDpIdx[k]], &ap[k], sizeof(float),
                    cudaMemcpyDeviceToDevice);
    }
    free(dpCPU);
    free(maxDpIdx);
    free(testMatrix);
    cublasFree(rx);
    cublasFree(dp);
    cublasFree(ap);
    cublasFree(v);
    cublasFree(mesMatrix);
    cublasFree(R);
    cublasFree(Q);
    return result;
}
```

Bibliografia

[1] E. J. Candes and T. Tao. Decoding by linear programming. *Ieee Transactions on Information Theory*, 51(12):4203–4215, 2005.

[2] IntelÂ® Corporation. IntelÂ® microprocessor export compliance metrics: IntelÂ® core i7-900 desktop processor series. IntelÂ® Support, 2011. `http://download.intel.com/support/processors/corei7/sb/core_i7-900_d.pdf`.

[3] NVIDIA Corporation. Tesla s1070 gpu computing system. CUDA Downloads, 2008. `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_S1070_US_Jun08_NV_LR_Final.pdf`.

[4] NVIDIA Corporation. Cuda downloads — nvidia developer zone. CUDA Downloads, 2011. `http://www.nvidia.com/getcuda`.

[5] NVIDIA Corporation. Cuda gpus — nvidia developer zone. CUDA GPUs, 2011. `http://developer.nvidia.com/cuda-gpus`.

[6] D. L. Donoho. Compressed sensing. *Ieee Transactions on Information Theory*, 52(4):1289–1306, 2006.

[7] National Science Foundation and Department of Energy. BLAS. http://www.netlib.org/blas/, 2010.

[8] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 1 edition, February 2010.

[9] J. Kruger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *Acm Transactions on Graphics*, 22(3):908–916, 2003.

[10] S. G. Mallat and Z. F. Zhang. Matching pursuits with time-frequency dictionaries. *Ieee Transactions on Signal Processing*, 41(12):3397–3415, 1993.

[11] Y. C. Pati, R. Rezaiifar, and P. S. Krishnaprasad. Orthogonal matching pursuit - recursive function approximation with applications to wavelet decomposition. *Conference Record of the Twenty-Seventh Asilomar Conference on Signals, Systems and Computers, Vols 1 and 2*, pages 40–44 1659, 1993.

[12] Gilbert Strang. *Linear Algebra and Its Applications*. Brooks Cole, February 1988.

[13] M. E. Davies T. Blumensath. On the difference between orthogonal matching pursuit and orthogonal least squares. *draft*, 2007.

[14] J. A. Tropp and A. C. Gilbert. Signal recovery from random measurements via orthogonal matching pursuit. *Ieee Transactions on Information Theory*, 53(12):4655–4666, 2007.

[15] Joel A. Tropp, Jason N. Laska, Marco F. Duarte, Justin K. Romberg, and Richard G. Baraniuk. Beyond nyquist: Efficient sampling of sparse bandlimited signals. *Ieee Transactions on Information Theory*, 56(1):25p, 2010. Vol. 56 Issue 1, p520-544 25p; 3 Diagrams, 5 Graphs.

[16] Y. Tsaig and D. L. Donoho. Extensions of compressed sensing. *Signal Processing*, 86(3):549–571, 2006.