Implementing a Side-Lobe Canceller Algorithm on GPU

Dr. farshad khunjush

Morteza Sadeghi – 9130202

هدف

پیاده سازی یکی از الگئریتم های side-lobe cancellation روی gpu میباشد. از میان الگوریتم های پیاده سازی SLC چند مورد را بررسی میکنیم و یکی را برای اجرا بر روی gpu پیاده و بهینه سازی میکنیم. در اینجا روش LMS پیاده سازی میشود.

SLC

Side-lobe Cancellation از دسته الگوریتم های حذف تداخلات است، یعنی تداخلاتی که بواسطه آنتن های دیگر یا jammer ها اتفاق میفتد را حذف میکند. فرض بر این است که این تداخلات، امواجی با پهنای باند فراتر از میانگین پنهای باند قابل دریافت آنتن میباشد و بنابراین این نویزها در side-lobe موج اصلی اتفاق میفتد، و SLC ها میخواهند این side-lobe ها را تا جای ممکن حذف کنند، این کار اولا نیاز به داشتن مدلی از سایر تداخلات ایجادشده توسط آنتن های محیط دارد، سپس یک SLC باید این تداخلات را از موج دریافتی تفریق کند ولی اینکار غیرممکن است و SLC ناچارا پارامترهای محیط را بصورت متغیرهای تصادفی مدل میکند. SLC برای کاهش تاثیر این تداخلات، با یک مساله بهینه سازی مواجه است.

این مساله بهینه سازی معمولا از طریق Least-Square-Mean یا LMS حل میشود، دراینصورت هرچقدر دقت بالاتر برود، زمان اجرا طولانی تر میشود. برای کاهش زمان اجرا، از بهینه سازی هایی استفاده میشود، مثلا داده ها را تبدیل فوریه میکنند. چند روش پیاده سازی عبارتند از LSM و RLS و SMI و HT که در ادامه بررسی میشوند.

همچنین وقتی تعداد آنتن ها بیشتر از یکی باشد، باید به تعداد آنتن ها حلقه های تودرتو داشته باشیم تا تداخلات همه آنتن ها ازبین برود، البته SLC هیچوقت نمیتواند این تداخلات را کاملا از بین ببرد و همچنین با افزایش تعداد آنتن ها و حلقه ها از قدرت SLC کاسته میشود.

از SLC بجز رادارها میتوان در هرجایی که از سیگنال ها نمونه برداری میشود هم استفاده کرد، مثل میکروفون، و اپلیکیشن هایی هستند که بکمک روشهای سیگنال فیلترینگ مثل SLC و GPU این کار را انجام میدهند.

1 - LSM (least square means)[1] and RLS (recursive least squares)[1]

هردو روش یک نوع برآورد آماری از نوع stochastic approximation of the steepest descent method هستند که سعی میکنند پارامترهای یک رابطه را طوری تنظیم کنند که آن رابطه مینیمم شود. LMS از روش گاوس نیوتون برای برآورد مینیمم استفاده میکند. روش های LMS چون هزینه محاسباتی کمتری دارند بیشتر استفاده میشوند.

Computation flow chart of the LMS GSC algorithm.

LMS GSC algorithm			
Eqn.	Function	CMUL	CADD
1	FOR $i = 1$ TO $K-1$ $\mathbf{x}_i(n) = [x_i(n)\ \ x_i(n-1)\ \ \ldots\ \ x_i(n-P)]^T$ $y_i(n) = \mathbf{w}_i^H(n-1)\mathbf{x}_i(n)$ END i	P	$P-1$
2	$y(n) = \sum_{i=1}^{K-1} y_i(n)$	0	$K-2$
3	$e(n) = d(n) - y(n)$	0	1
4	FOR $i = 1$ TO $K-1$ $\mathbf{w}_i(n) = \mathbf{w}_i(n-1) + \mu\mathbf{x}_i(n)e^*(n)$ END i	P	P
	TOTAL COST	$2(K-1)P$	$2(K-1)P$

چون پارامتر µ یک tradeoff بین سرعت همگرایی و جواب دقیق بوجود میاورد، نیاز به روشی سریعتر میباشد. روش هایی هم ارائه شده اند تا بازهم همگرایی LMS را بالا ببرند، مثلا تبدیل داده ورودی تحت تبدیل فوریه DFT باعث کاهش میزان محاسبات میشود:

Computation flow chart of the Transform Domain LMS GSC algorithm.

Transform Domain LMS GSC algorithm					
Eqn.	Function	CMUL	CADD	RMUL	RADD
	FOR $i = 1$ TO $K-1$				
1	$u_i(n) = x_i(n) - \rho^P x_i(n-P)$	0	1	0	0
	FOR $m = 0$ TO $P-1$				
2	$f_{i,m+1}(n) = \rho e^{-j\frac{2\pi m}{P}} f_{i,m+1}(n-1) + u_i(n)$	1	1	0	0
3	$p_{i,m+1}(n) = \lambda p_{i,m+1}(n-1) + (1-\lambda)\lvert f_{i,m+1}(n)\rvert^2$	0	0	2	2
4	$F_{i,m+1}(n) = \mu \frac{f_{i,m+1}(n)}{p_{i,m+1}(n)}$	0	0	2	0
	END m				
	$\mathbf{f}_i(n) = [f_1(n) \dots f_P(n)]^T$				
	$\mathbf{F}_i(n) = [F_1(n) \dots F_P(n)]^T$				
5	$y_i(n) = \mathbf{W}_i^H(n-1)\mathbf{f}_i(n)$	P	$P-1$	0	0
	END i				
6	$y(n) = \sum_{i=1}^{K-1} y_i(n)$	0	$K-2$	0	0
7	$e(n) = d(n) - y(n)$	0	1	0	0
	FOR $i = 1$ TO $K-1$				
8	$\mathbf{W}_i(n) = \mathbf{W}_i(n-1) + \mathbf{F}_i(n)e^*(n)$	P	P	0	0
	END i	0	0	0	0
	TOTAL COST	$3(K-1)P$	$(K-1)(3P+1)$	$4(K-1)P$	$2(K-1)P$

روش هایی برای افزایش بیشتر سرعت محاسبه ضرب اعداد مختلط هم ارائه شده تحت عنوان algorithmic strength reduction

Transform که تعداد محاسبات را کاهش میدهد. درنهایت هرچند تعداد ضرب ها کاهش میابد ولی تعداد جمع ها هم افزایش میابد ولی در حالت کلی محاسبات سبک تر میشود. مجموعه استفاده از این دو بهینه سازی تبدیل فوریه و SR تحت عنوان SR-DT-LMS ارائه شده که تعداد نهایی ضرب و جمع آن بصورت زیر است :

SR TD LMS GSC algorithm			
	Function	RMUL	RADD
	FOR $i = 1$ TO $K-1$		
	$\widehat{K}_m = \rho\left(\cos(\frac{2\pi m}{P}) - \sin(\frac{2\pi m}{P})\right), \quad \tilde{K}_m = \rho\left(\cos(\frac{2\pi m}{P}) + \sin(\frac{2\pi m}{P})\right)$		
1	$u_{\Re,i}(n) = x_{\Re,i}(n) - \rho^P x_{\Re,i}(n-P)$	0	1
2	$u_{\Im,i}(n) = x_{\Im,i}(n) - \rho^P x_{\Im,i}(n-P)$	0	1
	FOR $m = 0$ TO $P-1$		
3	$\widehat{f}_{i,m+1}(n) = f^{\Re}_{i,m+1}(n) - f^{\Im}_{i,m+1}(n)$	0	1
4	$f_{i,m+1,1}(n) = \tilde{K}_{m+1}\widehat{f}_{i,m+1}(n-1) + u_{\Re,i}(n)$	1	1
5	$f_{i,m+1,2}(n) = \rho\cos(\frac{2\pi m}{P})\widehat{f}_{i,m+1}(n-1) + u_{\Im,i}(n)$	1	1
6	$f_{i,m+1,3}(n) = \widehat{K}_m f^{\Im}_{i,m+1}(n-1)$	1	0
7	$f^{\Re}_{i,m+1}(n) = f_{i,m+1,1}(n) + f_{i,m+1,3}(n)$	0	1
8	$f^{\Im}_{i,m+1}(n) = f_{i,m+1,2}(n) - f_{i,m+1,3}(n)$	0	1
9	$p_{i,m+1}(n) = \lambda p_{i,m+1}(n-1) + (1-\lambda)\left((f^{\Re}_{i,m+1}(n))^2 + (f^{\Im}_{i,m+1}(n))^2\right)$	2	2
10	$F^{\Re}_{i,m+1}(n) = \frac{f^{\Re}_{i,m+1}(n)}{p_{i,m+1}(n)}, \quad F^{\Im}_{i,m+1}(n) = \frac{f^{\Im}_{i,m+1}(n)}{p_{i,m+1}(n)}$	2	0
11	$\widehat{F}^{\Re}_{i,m+1}(n) = F^{\Re}_{i,m+1}(n) - F^{\Im}_{i,m+1}(n)$	0	1
	END m		
	$\mathbf{f}_{\Re,i}(n) = [f^{\Re}_{i,1}(n)\ f^{\Re}_{i,2}(n) \ldots f^{\Re}_{i,P}(n)]^T$		
	$\mathbf{f}_{\Im,i}(n) = [f^{\Im}_{i,1}(n)\ f^{\Im}_{i,2}(n) \ldots f^{\Im}_{i,P}(n)]^T$		
	$\widehat{\mathbf{f}}_i(n) = [\widehat{f}_{i,1}(n)\ \widehat{f}_{i,2}(n) \ldots \widehat{f}_{i,P}(n)]^T$		
	$\mathbf{F}_{\Re,i}(n) = [F^{\Re}_{i,1}(n)\ F^{\Re}_{i,2}(n) \ldots F^{\Re}_{i,P}(n)]^T$		
	$\mathbf{F}_{\Im,i}(n) = [F^{\Im}_{i,1}(n)\ F^{\Im}_{i,2}(n) \ldots F^{\Im}_{i,P}(n)]^T$		
	$\widehat{\mathbf{F}}_i(n) = [\widehat{F}_{i,1}(n)\ \widehat{F}_{i,2}(n) \ldots \widehat{F}_{i,P}(n)]^T$		
12	$Y_{i,1}(n) = \mathbf{C}_i^T(n-1)\mathbf{f}_{\Re,i}(n)$	P	$P-1$
13	$Y_{i,2}(n) = \mathbf{D}_i^T(n-1)\mathbf{f}_{\Im,i}(n)$	P	$P-1$
14	$Y_{i,3}(n) = -\left(\mathbf{C}_i(n-1) + \mathbf{D}_i(n-1)\right)^T \widehat{\mathbf{f}}_i(n)$	P	$2P-1$
15	$y_{\Re,i}(n) = Y_{i,1}(n) + Y_{i,3}(n), \quad y_{\Im,i}(n) = Y_{i,2}(n) + Y_{i,3}(n)$	0	2
	END i		
16	$y_{\Re}(n) = \sum_{i=1}^{K-1} y_{\Re,i}(n), \quad y_{\Im}(n) = \sum_{i=1}^{K-1} y_{\Im,i}(n)$	0	$2(K-2)$
17	$e_{\Re}(n) = d_{\Re}(n) - y_{\Re}(n), \quad e_{\Im}(n) = d_{\Im}(n) - y_{\Im}(n)$	0	2
18	$\hat{e}(n) = e_{\Re}(n) - e_{\Im}(n)$	0	1
	FOR $i = 1$ TO $K-1$		
19	$\mathbf{G}_{i,1}(n) = 2e_{\Re}(n)\mathbf{F}_{\Re,i}(n), \quad \mathbf{G}_{i,2}(n) = 2e_{\Im}(n)\mathbf{F}_{\Re,i}(n)$	$2P$	0
20	$\mathbf{G}_{i,3}(n) = \hat{e}(n)\widehat{\mathbf{F}}_i(n)$	P	0
21	$\mathbf{C}_i(n) = \mathbf{C}_i(n-1) + \mu\left(\mathbf{G}_{i,1}(n) + \mathbf{G}_{i,3}(n)\right)$	0	$2P$
22	$\mathbf{D}_i(n) = \mathbf{D}_i(n-1) + \mu\left(\mathbf{G}_{i,2}(n) + \mathbf{G}_{i,3}(n)\right)$	0	$2P$
	END i		
	TOTAL COST	$13P(K-1)$	$16P(K-1) + 3(K-1) + 1$

نتیجه اینکه 20 درصد تعداد ضرب ها کاهش و 20 درصد تعداد جمع ها افزایش یافته است.

2 - **SMI** (sampled matrix inversion)

این روش برای حل مساله بهینه سازی در SLC زیاد استفاده شده، و روشی است که سرعت همگرایی آن به جواب، مستقل از حجم نویزهای ورودی است، حتی این روش قابلیت اجرا روی GPU را نیز داراست و مقالاتی در این زمینه موجود است. ولی عیب این روش در زمانی است که نویز بالا باشد، بنابراین این روشی است که فقط در شرایط خاصی بدرد میخورد.

3 - **HT** (householder transform) [3]

این تبدیل برای زمانی مناسب است که بین سیگنال های نویز و اصلی، سباهت زیادی وجود داشته باشد، این روش در اینصورت قادر است نویز را بخوبی جدا کند. یک نمونه از الگوریتم های این روش HCQN است که در زیر شبه کد آن آمده است :

Table 1: The HCQN Algorithm

Initialization: α, $\mathbf{R}_{H(0)}^{-1}$ and $\overline{\mathbf{W}}_0 = \mathbf{QC}\left(\mathbf{C}^H\mathbf{C}\right)^{-1}\mathbf{f}$

For each k

$\overline{\mathbf{X}}_k = \mathbf{QX}_k$; according to an efficient procedure

$\hat{\mathbf{X}}_k$ = first p elements of $\overline{\mathbf{X}}_k$;

$\mathbf{X}_{H(k)}$ = last MN-p elements of $\overline{\mathbf{X}}_k$;

$$\overline{\mathbf{W}}_{k-1} = \left[\frac{\hat{\mathbf{W}}_0}{-\mathbf{W}_{H(k-1)}}\right];$$

$$e_k = \hat{\mathbf{W}}_0^H \hat{\mathbf{X}}_k - \mathbf{W}_{H(k)}^H \mathbf{X}_{H(k)};$$

% Update of the coefficient error according

% to the unconstrained algorithm:

$$\mathbf{t}_k = \mathbf{R}_{H(k-1)}^{-1} \mathbf{X}_{H(k)};$$

$$\tau_k = \mathbf{X}_{H(k)}^H \mathbf{t}_K;$$

$$\mu_K = \frac{1}{2\tau_k};$$

$$\mathbf{R}_{H(k)}^{-1} = \mathbf{R}_{H(k-1)}^{-1} + \frac{\mu_k - 1}{\tau_K} \mathbf{t}_k \mathbf{t}_k^H;$$

$$\mathbf{W}_{H(k)} = \mathbf{W}_{H(k-1)} + \alpha \frac{e_k}{\tau_k} \mathbf{t}_k;$$

End

در تصویر زیر نشان داده میشود که استفاده از تبدیل HT در روش های LMS و NLMS و QN حجم محاسبات را تا اندازه بسیار زیادی کاهش داده است :

Computational Complexity

ALG.	ADD.	MULT.	DIV.
CLMS	$4MN - 2$	$4MN + 1$	0
GSC LMS	$(MN)^2 + MN - 2$	$(MN)^2 + 2MN - 1$	0
HCLMS	$4MN - 3$	$4MN$	0
NCLMS	$6MN - 3$	$6MN + 1$	1
GSC NLMS	$(MN)^2 + 2MN - 4$	$(MN)^2 + 3MN - 2$	1
NHCLMS	$5MN - 5$	$5MN - 1$	1
CQN	$(MN)^3 + 7(MN)^2 + 9MN - 8$	$(MN)^3 + 8(MN)^2 + 12MN + 1$	$MN + 4$
GSC QN	$3(MN)^2 - 4$	$3(MN)^2 - MN$	3
HCQN	$2(MN)^2 - MN - 4$	$2(MN)^2$	3

پیاده سازی

از میان روش های یادشده، روش LMS کاربرد بیشتری داشته است و قابلیت اجرای بر روی gpu را هم دارد، همچنین نمونه کد آن نیز فراوان یافت میشود، بنابراین از این روش استفاده میکنیم. در نمونه کد زیر، این روش در متلب پیاده سازی شده است، این کد از آن دسته کدهایی است که از توابع خاص Matlab استفاده نکرده و بنابراین مهاجرت آن به زبان C++ آسان است. هدف من این است که آنرا به زبان c برده و روی GPU بهینه کنم :

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%ÔÚ¼¸ÖÖ²»Í¬µÄÐÅÔë±ÈÏÂ¼ÆËãÎóÂëÂÊ
for SNR=-6:2:18
    %SNR=30;
%diBi=1/(2*SNR)^0.5;
%SNR=10;
diBi=1/(2*10^(SNR/10))^0.5;
Noise1=Noise1*diBi;%%%%¼ÓÈëÐÅÔë±È

x_train=xxx(:,(1:Len_train));
N_train=Noise1(:,(1:Len_train));

x=B*x_train+N_train; %¼ÓÔëÉù
x1=B*xxx+Noise1; %¼ÓÔëÉù
R=x*x';
W=inv(R)*B*inv((B'*inv(R)*B))*e;

y=W'*x1;
output=real(y);
d_output=output;
d_output(find(d_output>=0))=1;
d_output(find(d_output<0))=-1;
d=xxx(1,:);
%error=norm(d-d_output,1)/2;

error=length(find(d-d_output))
pe=error/Len;
Plot_Pe=[Plot_Pe pe];

end;
```

فایل lcmv.mat

برای پیاده سازی راحت، لازم است که یک کلاس برای Matrix ایجاد کنم، استفاده از کلاس این مزیت را دارد که کد نوشته شده را خوانا کند و مدیریت آنرا بهتر کند، هرچند ممکن است سربار عملیاتی را افزایش دهد. برای مثال تابع transpose وقتی روی

GPU برده میشود، زمان اجرا را کاهش نداده و افزایش میدهد، علت آن است که در هر تابعی از کلاس matrix که از GPU استفاده میکند، علاوه بر انتقال آرایه به device و بالعکس، باید یک شئ جدید هم بعنوان جواب ساخته شود، و ساخته شدن آنهم نیازمند کپی مجدد آرایه است. این اجبار برای ساختن شئ جدید، مخصوصا بدلیل استفاده از operation overload در C++ تحمیل میشود. به این دلیل(بنظر من) در کلاس matrix ، تابع transpose برای اجرا روی GPU بصرفه نیست. درحالیکه عملیات ضرب و جمع برای اجرا روی gpu بصرفه هستند و زمان اجرا را به یک سوم کاهش میدهند.

عملگرهای ضرب و جمع ماتریس برای اجرا روی GPU بهینه شده اند. برای پیاده سازی تابع ضرب از مثال NVidia استفاده کردم، و برای عملگر inversion از کتابخانه BoInc کمک گرفتم.

در پوشه sidelobe_cancellation، فایل report.txt زمان اجرا برای تغییرات اعمال شده را نشان میدهد. در پوشه ی اول بنام serial_1 حاوی کد اجرا روی cpu میباشد. پوشه دوم بنام gpu_transpose_2 شامل پیاده سازی gpu برای تابع transpose میباشد و در فایل های transpose.cu و transpose.h این تابع پیاده سازی شده است که از داخل فایل matrix.cu فراخوانی میشوند. برای این پیاده سازی سه مدل naive و coalesced و no-bank-conflicts استفاده شده، که هیچ کدام از اینها سریعتر از cpu اجرا نشدند. در پوشه سوم بنام gpu_multiply_3 توابع ضرب در فایل multiply.cu پیاده سازی شده اند. استفاده از این توابع سرعت اجرا را به یک سوم کاهش داده. در پوشه چهارم بنام gpu_add_4 توابع ضرب و جمع باهم در فایل MatrixGpu.cu پیاده سازی شده اند، در اینجا دو فایل اجرایی هستند که اولی مربوط به سربارگذاری عملگر + است و دومی مربوط به سربارگذاری عملگر += که مورد دوم زمان اجرای بیشتری برای اجرا مصرف کرد، و علت این را نمیدانم. در پوشه پنجم عملگر inversion پیاده سازی شده که خطای segmentation fault آنرا نتوانستم تشخیص بدهم.

file	execution time (milli-secconds)
0_lcmv_serial	9830
1_lcmv_transpose_naive	12485
2_lcmv_transpose_coalesced	12487
3_lcmv_transpose_NoBankConflict	12863
4_lcmv_NVidiaMultiply	3627
5_lcmv_NVidiaMultiply	3472
6_lcmv_mult_add	3430
7_lcmv_mult_add	3980

فایل report.txt ، زمان اجرا برای پیاده سازی ها موفق را نشان میدهد. ، فایل های اجرایی با همین نام ها در پوشه ها وجود دارند.

نتیجه

باتوجه به این گزارش، برنامه sidelobe-cancellation میتواند روی GPU با سرعت چندبرابر اجرا شود، به این دلیل که حداقل از عمل ضرب ماتریس بسیار استفاده میکند که همین سرعت را بسیار افزایش خواهد داد.

منابع :

[1] - **IMPLEMENTATION OF ADAPTIVE GENERALIZED SIDELOBE CANCELLERS USING EFFICIENT COMPLEX VALUED ARITHMETIC -** GEORGE-OTHON GLENTIS

[2] - **On the Use of Householder Transformation in Adaptive Microphone Array - César A. Medina Carlos V. Rodríguez José A. Apolinário Jr.**