Procesamiento de Señales SAR: Algoritmo RDA para GPGPU

Mónica Denham[1,2,3], Javier Areta[1,2], Isidoro Vaquila[2,5], Fernando G. Tinetti[3,4]

[1]Ingeniería Electrónica, Sede Andina, Universidad Nacional de Río Negro
[2]Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) Argentina
[3]Instituto de Investigación en Informática LIDI. Fac. de Informática – UNLP
[4]Comisión de Investigaciones Científicas de la Provincia de Buenos Aires (CIC)
[5]INVAP
{mdenham,jareta}@unrn.edu.ar, ivaquila@invap.com.ar,
fernando@lidi.unlp.edu.ar

Resumen En este trabajo se presenta una solución secuencial y una paralela del algoritmo RDA (*Range Doppler Algorithm*) para el procesamiento de señales de radares SAR (*Synthetic Aperture Radar*). La solución paralela se desarrolló en C CUDA para GP-GPU (*General Purpose Graphic Processing Units*). Se describe la solución desarrollada, se muestran los primeros resultados y se describen las futuras optimizaciones para dicho algoritmo.

Keywords: Procesamiento Paralelo, HPC, CUDA, GP-GPU, Procesamiento de señales, radares SAR.

1. Introducción

Los radares de apertura sintética (SAR) son radares de pequeñas dimensiones que se acoplan a aeronaves (aviones o satélites) y aprovechan el desplazamiento de dicha aeronave para obtener imágenes de alta resolución. Se utilizan para formar imágenes de la superficie terrestre, detectar objetivos, realizar el seguimiento de objetivos móviles, etc.

En este trabajo la información recolectada y almacenada por el radar es procesada para formar imágenes de la superficie terrestre.

El modo de operación de estos radares se basa en que el radar avanza con la trayectoria de la aeronave mientras envía sucesivos pulsos y almacena sus ecos. Toda la información almacenada se sintentiza para formar una única imagen [3] [4] [9] [11] [12].

Debido a la frecuencia de envío de pulsos y al muestreo de los ecos recibidos, los datos recolectados son almacenados en arreglos de 2 dimensiones: las filas corresponden a cada pulso enviado (dimensión llamada acimut) y cada eco es muestreado y almacenado en una fila (el muestreo en rango define las columnas, dimensión llamada rango). Debido a las altas frecuencias con que se opera (en

rango y acimut), las matrices suelen ser de grandes dimensiones, normalmente almacenan millones de datos. A estos datos se los conoce como *datos crudos*.

En la actualidad, existen diversos algoritmos para el procesamiento de señales SAR [4] [6] [9] [11] [12]. Los algoritmos más conocidos y utilizados son: *Range Doppler Algorithm* (RDA), *Chirp Scaling Algorithm* (CSA), *Omega-K Algorithm*, *Back-projection Algorithm*, etc.

Dichos algoritmos se basan en aplicar filtros sobre los datos, transformadas de Fourier, antitransformadas, etc., todas operaciones con altos costos computacionales. Como se especificó anteriormente, estos algoritmos operan sobre una gran cantidad de datos. Estas características obligan a desarrollar soluciones desde la tecnología HPC (*High Performance Computing*). Además, es frecuente que este procesamiento esté ligado a aplicaciones de tiempo real haciendo aún más necesaria la implementación de algoritmos con bajos tiempos de respuesta, sin perder calidad de las imágenes generadas.

En este trabajo se utilizan procesadores gráficos de tipo GPGPU como arquitectura de ejecución paralela. Estas son placas que nacen para procesamiento gráfico (en el mercado de video juegos) pero su alto poder de cómputo, alto rendimiento y bajo costo ha hecho que se desarrollen placas gráficas de uso general (GPGPU: *General Purpose Graphic Processing Unit*).

Este trabajo es la continuación de [7], donde se enumeran las principales características del procesamiento de datos crudos SAR, como así también los pasos de los algoritmos RDA y CSA. Además, se muestran los rasgos más importantes de las arquitecturas GPGPU, poniendo énfasis en la organización de los componentes de las placas gráficas, la jerarquía de *threads* y jerarquía de memorias. Además se introduce las principales características de la programación en C CUDA.

En las siguientes secciones se presentan los aspectos principales de la solución secuencial y paralela del algoritmo RDA. La solución paralela es una primera aproximación en la que aún no se consideran aspectos más avanzados del hardware [2] [5]. Luego se presentan resultados obtenidos con dichos algoritmos y una comparación y análisis de los mismos. Además se plantean los pasos futuros que corresponden con optimizaciones del algoritmo.

1.1. Procesamiento de Señales SAR: Range Doppler Algorithm

El algoritmo RDA fue desarrollado en 1978 para procesar datos del primer radar SAR (SEASAT SAR) y hasta la actualidad es uno de los algoritmos más utilizados. Su principal característica es que usa operaciones en el dominio de la frecuencia en ambas dimensiones (rango y acimut), operando de esta forma en 1 dimensión, logrando mayor simplicidad [4].

El algoritmo se basa en la aplicación de tres operaciones: compresión en rango, RCMC (*Range Cell Migration Correction*), y compresión en acimut. Con estas operaciones se enfocan los datos en rango, luego se corrige migración de celdas en rango y por último se comprimen los datos en acimut para enfocar los datos en esta segunda dimensión.

La compresión de rango se lleva a cabo realizando una convolución rápida en cada fila de la matriz: se realiza una FFT para llevar los datos al dominio de la frecuencia, se multiplica por un filtro y por último se realiza una IFFT para llevar los datos al dominio del tiempo nuevamente.

Asumiendo el caso de radares transportados en aviones, donde las distancias son cortas, se puede asumir que la superficie terrestre es plana. Considerando un objetivo puntual, es facil observar que a medida que avanza el radar en su recorrido, la distancia del radar a dicho objetivo cambia con el tiempo (acimut). Esta diferencia en la distancia en función del tiempo en acimut genera migración de celdas en rango en los datos crudos. Es necesario entonces corregir esta migración de celdas. Dicha corrección se lleva a cabo mediante un corrimiento de celdas en función de la migración producida.

Por último, se realiza la compresión en acimut: cada columna de la matriz se pasa al dominio de la frecuencia (aplicando una FFT), se aplica un filtro, y se realiza una antitransformada para llevar los datos al dominio del tiempo.

2. RDA en C y C CUDA

Como punto de partida se ha desarrollado dicho algoritmo en MATLAB. Esta primer implementación aportó claridad y experiencia en las distintas operaciones del algoritmo y sus posibles implementaciones.

Además, dicha implementación permite verificar la correctitud de las operaciones a cada paso, haciendo que el desarrollo en C y C CUDA se desarrollen con plena seguridad.

El algoritmo secuencial en el lenguaje C se desarrolló con el objetivo de tener una métrica con la cual poder comparar el rendimiento del algoritmo paralelo.

Particularmente, se ha trabajado con una librería concreta para las operaciones de FFT e IFFT desarrollada por Mark Borgerding. Esta librería implementa de forma rápida las FFT e IFFT y se puede utilizar con datos de simple o doble precisión (`http://sourceforge.net/projects/kissfft/`).

Además de los algoritmos secuenciales se ha desarrollado el algoritmo en paralelo, especialmente diseñado e implementado para su ejecución en GPGPU.

Paralelización de RDA

Actualmente se dispone de una primer versión de RDA en C CUDA, la cual será explicada y evaluada. Este primer desarrollo de la solución paralela no se considera que sea óptima, pero constituye un punto de partida para llegar a soluciones más eficientes.

En el corto plazo, se buscarán modificiones al algoritmo paralelo para, principalmente, realizar una distribución de trabajo eficiente y una utilización de la jerarquía de memorias que logre rendimiento óptimo.

Como ya se ha expuesto, muchas de las operaciones de RDA se basan en aplicación de filtros, convoluciones, FFT, IFFTs. La paralelización del algoritmo se basa en: las operaciones FFT e IFFT se resuelven utilizando las operaciones

paralelas de la librería `cuFFT` y las aplicaciones de filtros, convoluciones, normalizaciones, se paralelizan creando bloques de *threads* y haciendo que cada *thread* compute un único elemento del arreglo (señal).

Libreria *cuFFT* [8] es una librería desarrollada por NVIDIA [1] que resuelve las Transformadas Rápidas de Fourier (y antitransformadas). Las operaciones en esta librería se implementan con la estrategia *divide y vencerás* y logran algoritmos eficientes para conjuntos de valores reales y complejos. La librería `cuFFT` provee interfaces simples que permiten al usuario hacer uso de la potencia de cálculo y poder de paralelismo de las placas GPU, de forma transparente.

2.1. Compresión en Rango

Compresión en rango secuencial Esta operación se desarrolla de forma secuencial, fila por fila, realizando una convolución rápida: FFT, aplicación de filtro, IFFT. Como ya se ha expuesto, las operaciones FFT e IFFT se resuelven de forma secuencial con operaciones de la librería KISS_FFT. A su vez, el filtrado de la señal se resuelve con un producto punto también secuencial. A continuación se muestra un pseudocódigo simplificado de esta operación.

Compresión de Rango Secuencial

```
...
for (cada fila)
{
    FFT;
    producto punto;
    IFFT;
}
```

La transformada discreta de Fourier (TDF) tiene orden de complejidad $O(n^2)$. A su vez, las implementaciones FFT (*Fast Fourier Transform*) son algoritmos recursivos que siguen la estrategia *divide y vencerás*, y pueden llegar a reducir este orden de complejidad a $O(n \log n)$. El producto punto tiene orden de complejidad $O(n)$. Estas operaciones se ejecutan por cada una de las filas, llegando a un orden de complejidad que $O(n^2 \log n)$.

Teniendo en cuenta las dimensiones de las matrices de datos crudos, decrementar este orden de complejidad es un requisito fundamental para mejorar el rendimiento de esta aplicación.

Compresión en rango paralelo En esta primer propuesta paralela, el procesamiento también se realiza por cada una de las filas de la matriz: las operaciones FFT e IFFT se realizan utilizando operaciones de la librería *cuFFT* de CUDA. Como ya se ha dicho, las operaciones de dicha librería están optimizadas para obtener máximo rendimiento en GPU.

Luego de la FFT, se realiza un producto punto de toda la fila. Para dicha operación se implementa un *kernel* el cual se ejecuta con tantos *threads* como elementos tenga la fila. Todos los *threads* se ejecutan de forma concurrente (con un elevado paralelismo) y cada uno resuelve un único producto y lo almacena en un lugar de la fila resultante (figura 1).

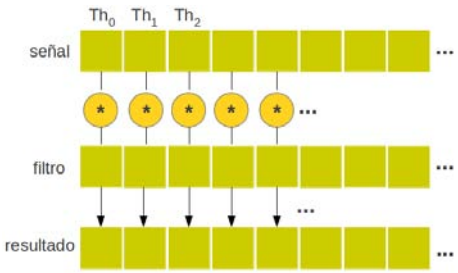

Figura 1. Kernel CUDA: cada *thread* calcula el producto de un elemento del vector resultante.

Una de las características de la utilización de GPUs y CUDA es que el programador puede crear un número de *threads* muy elevado, asumiendo que no hay restricción máxima de cantidad de *threads*. Por esto mismo, crear un *thread* por cada elemento de la fila, que compute un único producto es natural, independientemente de las grandes dimensiones que pueda tener la estructura de datos con la que se opera. De esta forma, todos estos productos se resuelven "de forma simultánea" y este es un de los aspectos principales de las placas gráficas.

Como se mencionó anteriormente, esta primer propuesta paralela realiza estas operaciones por cada una de las filas. A futuro, se preve implementar el procesamiento de todas las filas a la vez, aprovechando la ausencia de dependencia entre los datos y su procesamiento.

2.2. Range Cell Migration Correction (RCMC)

Por cada fila de la matriz se calcula la cantidad de celdas enteras a corregir debido a la migración. Además se calcula la migración a nivel de fracción de celda, como se propone en [4].

RCMC Secuencial Por cada fila, se calcula la cantidad de celdas que el objetivo haya migrado en los datos crudos, dependiendo del tiempo en acimut. Luego se realiza un corrimiento de todos los elementos de la fila en función del cálculo previo.

A su vez, se ajusta la migracion fraccionaria utilizando un kernel de interpolación propuesto en [4]. La fila se convoluciona con dicho kernel para realizar un ajuste más preciso. A continuación se muestra un pseudocódigo de RCMC.

Corrección de migración de celdas en rango secuencial

```
...
for (cada fila)
{
    cálculo de migración de celdas enteras;
    cálculo de migración de fracción de celdas;

     /* corrección de migración de celdas */
    corrimiento de toda la fila;
    convolución con el kernel de interpolación;
}
```

RCMC Paralelo A nivel de celda entera se realiza el corrimiento con un *kernel* paralelo: cada uno de los *threads* copia un dato en la fila teniendo en cuenta el corrimiento. Se copian en paralelo todos los datos de la fila, evitando una iteración sobre la misma.

La corrección de migración de celdas a nivel de fracción de celdas se ejecuta en paralelo teniendo un *thread* por elemento de la fila resultado y cada *thread* realiza la multiplicación y suma correspondiente. Nuevamente, se evita iterar sobre los elementos de la fila.

De forma similar a la operación anterior, se observa que una posible mejora es implementar todos los corrimientos de todas las filas de forma concurrente. Esto evita iterar sobre cada una de las filas.

2.3. Compresión en Acimut

Por cada una de las columnas de la matriz resultante de las dos operaciones anteriores se realiza: FFT de la columna, producto punto entre la columna y el filtro (el mismo se encuentra en dominio de la frecuencia) y por último IFFT del resultado, para llevar los datos resultantes al dominio del tiempo.

Compresión en Acimut secuencial Esta operación es similar a la compresión de rango, pero se trabaja columna a columna. Se trabaja en el dominio de la frecuencia, por lo que el filtro se transforma al comienzo del procesamiento, y cada columna se transforma antes del producto.

Compresión en acimut secuencial

```
...
for (cada columna)
{
    FFT columna
    producto punto columna y filtro
    IFFT resultado
}
```

Compresión en Acimut paralelo Nuevamente, esta operación aprovecha las operaciones de la librería cuFFT, las cuales garantizan operaciones eficientes para resolver las transformadas (antitransformadas) de Fourier. Por otro lado, el producto punto se resuelve en paralelo, utilizando un *thread* por cada elemento del vector.

Esto se realiza de forma secuencial sobre cada una de las columnas de la matriz, lo cual se prevee modificar para realizar de forma concurrente todas las columnas (no existe dependencia de datos).

En las próximas secciones se mostrarán y analizarán los primeros resultados obtenidos con ambas implementaciones.

3. Resultados

Se realizará la comparación de rendimiento del código secuencial y el rendimiento de la primer implementación paralela.

La arquitectura utilizada está compuesta por un host multicore de 4 procesadores Intel(R) Core i5-2500 CPU @ 3.30GHz, con 7.7GB de memoria y sistema operativo Ubuntu de 64 bits. En dicha CPU se encuentra conectada una placa gráfica de tipo GeForce GTX 550Ti con 192 cores CUDA (dispuestos en 4 multiproceadores) de fabricante NVIDIA.

Se ha experimentado con 3 imágenes: un objetivo puntual (imagen sintética: un único objetivo en el medio de la imagen), y 2 fotos reales, una de las cuales es la vista de un aeropuerto y otra de un crater de un volcán.

Para disponer de los datos crudos de dichas imágenes se ha utilizado un simulador que simula el proceso de generación de los datos crudos. Dicho simulador fue desarrollado por uno de los miembros del equipo de trabajo.

Como ejemplo, la figura 2 muestra la foto del aeropuerto utilizada y la imagen obtenida luego de procesar los datos crudos con el algoritmo RDA implementado.

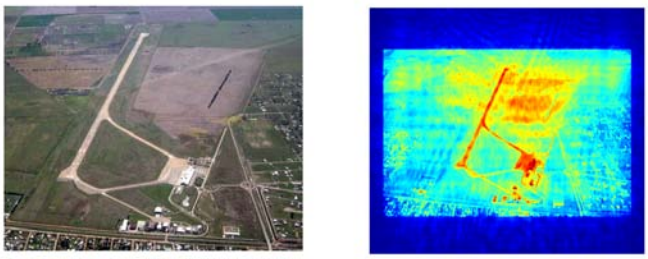

Figura 2. Imagen de aeropuerto, foto aérea e imagen obtenida con el algoritmo RDA.

Para los 3 casos de experimentación, se han tenido en cuenta los siguientes valores: el tiempo de exposición del objetivo es 3.4 segundos, y se trabaja a una frecuencia de envío de pulso de 600Hz. En total se toman 2000 pulsos.

Por otro lado, el eco del pulso recibido se muestrea para su almacenamiento y posterior procesamiento (valores discretos). La frecuencia de muestreo es de 120MHz. Debido a la distancia que se desea cubrir en rango, y al resto de valores propios de la geometría SAR, se almacenan 8000 datos en rango. Esto es, el eco recibido se muestrea y se obtienen 8000 valores. Estos valores definen matrices de 2000 filas de 8000 muestras (columnas). Esto corresponde a la matriz de datos crudos, matrices intermedias e imagen enfocada (imagen final).

Como el algoritmo RDA está bien definidos en 3 operaciones bien visibles, los primeros análisis se han realizado sobre cada una de estas operaciones. Se obtuvieron los tiempos de ejecución de las operaciones: compresión en rango, RCMC y compresión en acimut.

Los tiempos obtenidos se muestran en el cuadro 1 para las 3 operaciones del algoritmo RDA (en segundos). Se puede observar que los tiempos son muy similares para las 3 imágenes, por lo que se expondrán dichos tiempos en figuras sólo para un caso, valiendo el análisis para el resto. En dicho cuadro, RC es compresión en rango, RCMC es corrección de migración de celdas en rango y AC es compresión en acimut. Por cada imagen se muestra los tiempos secuenciales (Sec) y los tiempos del algoritmo paralelo (Par).

	Objetivo Puntual		Aeropuerto		Volcan	
	Sec	Par	Sec	Par	Sec	Par
RC	97.32	4.88	97.20	4.92	97.35	4.91
RCMC	0.57	0.04	0.56	0.04	0.57	0.04
AC	4.64	0.68	4.65	0.68	4.65	0.68

Cuadro 1. Tiempos obtenidos para las 3 imágenes (en segundos).

Para continuar con el análisis, la figura 3 muestra los tiempos del algoritmo RDA con una de las imágenes (objetivo puntual).

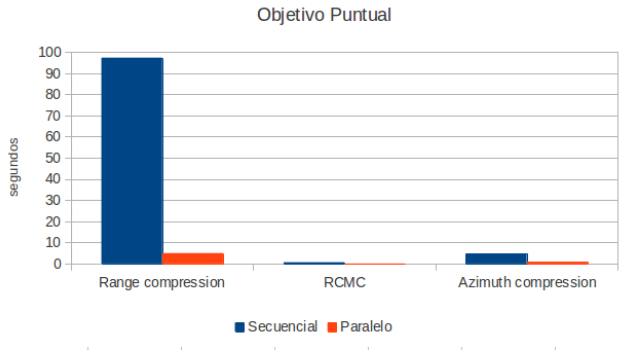

Figura 3. Tiempos de ejecución del algoritmo RDA con la imagen del objetivo puntual.

En dicha figura se puede distinguir con exactitud la ganancia de paralelizar la compresión en rango, pero las otras dos operaciones no se puede determinar debido a la escala del eje Y (tiempos en segundos). Por esto se muestran en la figura 4 estas operaciones en gráficos distintos, para poder comparar de forma precisa la diferencia de tiempos del algoritmo secuencial y paralelo. Considerar el cambio de escala del eje Y para dichos gráficos.

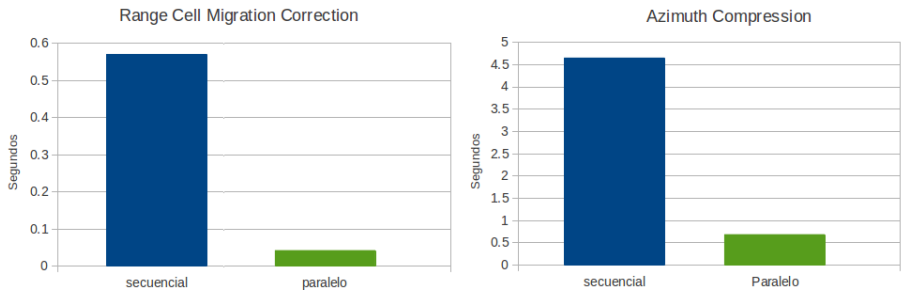

Figura 4. Tiempos de las operaciones RCMC y compresión en acimut en detalle.

En todos los casos es posible observar que la compresión en rango es la operación que más tiempo requiere cuando se resuelve de forma secuencial. Dicha operación resuelve una FFT, un producto punto y una IFFT por cada una de las filas. Debido a la operación FFT que se realiza, se realiza *zero padding* a los datos (filas), por esto se trabaja con 16000 valores en rango. Al finalizar esta operación, se vuelven a seleccionar los datos centrales de la antitransformada, que es donde se concentra la información útil (y se sigue el procesamiento usando 8000 muestras en rango).

Lo anteriormente dicho no sucede en las operaciones de corrección de celdas (se trabaja con filas de 8000 complejos) ni en compresión en acimut (columnas de 2000 complejos).

Tomando los tiempos secuenciales de cada operación como referencia, para la compresión de rango se observa un 95 % de reducción en el tiempo, en la operación RCMC un 93 % de reducción y en la compresión en acimut se observa un 86 % de reducción (aproximadamente).

Como se ha mencionado anteriormente, la solución paralela propuesta es una primer aproximación, para la cual no se consideran aspectos fundamentales en la optimización de códigos CUDA: cantidad y jerarquía de *threads* en cada kernel, movimiento de datos a memorias de más rápido acceso (local, compartida, textura o constante) [2] [5]. El próximo paso a seguir en esta línea es modificar estos aspectos y así lograr algoritmos paralelos más eficientes.

4. Conclusiones

Este trabajo presenta los primeros resultados del algoritmo RDA secuencial y paralelo utilizando C CUDA en GPGPU.

El algoritmo RDA cumple con los requerimientos necesarios para que su implementación y ejecución en GPU sea conveniente: alta carga computacional, independencia de datos, mínima transferencia de datos entre CPU y GPU (solo al comienzo y al final del procesamiento), no existen secciones críticas, los datos se mantienen en matrices logrando mapear las estructuras de datos con la disposición de los *threads* en los grids en cada kernel.

Los primeros resultados obtenidos muestran que la paralelización del algoritmo logra reducir significativamente los tiempos en las 3 operaciones del algoritmo RDA. Esto muestra que la paralelización es efectiva, y, teniendo en cuenta posibles optimizaciones, el algoritmo RDA promete un muy buen rendimiento en GPGPU. A corto plazo se espera optimizar estas tres operaciones para obtener algoritmos más eficientes aún.

Este trabajo muestra un primer paso abordando la comparación del algoritmo secuencial y del algoritmo paralelo. Como trabajo futuro también se estima la posibilidad de evaluar algoritmos propuestos en la literatura, con el fin de realizar comparaciones entre distintos algoritmos paralelos.

Referencias

1. NVIDIA home page, `http://la.nvidia.com/page/home.html`
2. Farber, R.: CUDA Application Design and Development. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2011)
3. Hein, A.: Processing of SAR Data. Springer (2004)
4. Ian G. Cumming, F.H.W.: Digital Processing of Synthetic Aperture Radar Data. Artech House (2005)
5. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2010)
6. Meier, O.F..E.H., Nüesch, D.R.: Processing sar data of rugged terrain by time-domain back-projection. SPIE 5980, SAR Image Analysis, Modeling, and Techniques VII, 598007 (2005), `http://dx.doi.org/10.1117/12.627647`
7. Denham M., Areta J.: Procesamiento de señales sar en GPGPU. In: CACIC 2012 (2012), `http://sedici.unlp.edu.ar/handle/10915/23633`
8. NVIDIA: CUDA Toolkit cuFFT Library. Programming Guide (March 2011)
9. Richards, M.A.: Fundamentals of Radar Signal Processing. McGraw-Hill (2005)
10. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, 1st edn. (2010)
11. Skolnik, M.I.: RADAR Systems. Tata McGraw-Hill
12. Soumekh, M.: Synthetic Aperture Radar Signal Processing with MATLAB Algorithms. Wiley-Interscience (1999)