Lámpsakos | N°. 14 | pp. 18-24 | julio-diciembre | 2015 | ISSN: 2145-4086 | Medellín–Colombia

Paralelización del Algoritmo Criptográfico GOST Empleando el Paradigma de Memoria Compartida

GOST Cryptographic Algorithm Parallelization Using Shared Memory Paradigm

Marlis Fulgueira-Camilo, Ing.
Centro de Investigación Tecnológica Integrada
La Habana, Cuba,
mfulgueirac@citi.cu

Omar A. Hernández-Duany, MSc.
Centro de Investigación Tecnológica Integrada
La Habana, Cuba,
ohernadez@citi.cu

Venus Henry-Fuenteseca, Ing.
Instituto Superior Politécnico José Antonio Echeverría
La Habana, Cuba,
vhenry@ceis.cujae.edu.cu

(Recibido el 15-05-2015. Aprobado el 20-06-2015)

Citación de artículo, estilo IEEE:
M. Fulgueira-Camilo, O. Hernández-Duany, V. Henry-Fuenteseca, "Paralelización del Algoritmo Criptográfico GOST Empleando el Paradigma de Memoria Compartida", Lámpsakos, N° 14, pp. 18-24, 2015.

Resumen. El artículo refiere el proceso de paralelización del algoritmo criptográfico GOST. La investigación realizada persigue como objetivo, reducir el tiempo de ejecución del algoritmo. El estudio no se encuentra enfocado al análisis de fortaleza del algoritmo criptográfico; donde se hace énfasis es en el método empleado para disminuir el tiempo de ejecución de los procesos cifre y descifre. Para ello se realiza un diseño paralelo basado en la metodología de Lan Foster, el cual es aplicado a dos implementaciones usando técnicas como: OpenMP y CUDA. Las comparaciones realizadas teniendo en cuenta, tanto al algoritmo secuencial como las implementaciones paralelas, demuestran una significativa reducción de tiempo, sin importar la técnica empleada. El mejor resultado se logra empleando CUDA.

Palabras clave: Computación Paralela y Distribuida; criptografía; CUDA; GOST

Abstract. The paper refers to the process of the cryptographic algorithm parallelization GOST. The aim pursued research, reduce the execution time of the algorithm. The study is not focused on the analysis of the strength of cryptographic algorithm, where the emphasis is on the method used is to reduce the execution time of the encrypt and decrypt processes. OpenMP, CUDA and OpenCL: To do a parallel design methodology based on Ian Foster, which is applied to three implementations using techniques like is performed. Comparisons considering both the sequential and parallel implementations algorithm, demonstrate a significant reduction of time, regardless of the technique used. The best result is achieved using CUDA.

Keywords: Cryptographic, CUDA, GOST, Parallel and Distributed Computation.

1. INTRODUCCIÓN

Un algoritmo criptográfico se caracteriza por convertir un texto claro, en otro, llamado texto cifrado. El contenido de la información es igual al anterior pero solo puede ser entendido por la persona autorizada[1]. Los canales de comunicación se suponen inseguros cuando se desea enviar información confidencial a través de estos, en cuyos casos se requiere cifrar el mensaje. El presente trabajo se enfoca en la reducción del tiempo de ejecución de la implementación del algoritmo criptográfico GOST (Estándar Gubernamental de la Unión de Repúblicas Socialistas Soviéticas). Los detalles del algoritmo fueron publicados en 1990 bajo el nombre de Estándar Soviético (GOST 28147-89). El estándar provee un nivel de seguridad flexible y puede ser empleado para proteger información en sistemas de cómputo y en redes de computadoras [2, 3].

La investigación tiene como propósito introducir técnicas de programación paralela que permitan mejorar el rendimiento del algoritmo en cuanto a su tiempo de ejecución. La versión paralela del algoritmo, se empleará para proteger la información que se envíe por un canal de comunicación. Es necesario destacar que los flujos de datos en los procesos de las redes, son crecientes con una tasa de transferencia elevada. La información que se transfiera por la red debe cifrarse en tiempo real y para lograr esto se hace necesario reducir el tiempo de ejecución del algoritmo.

Para darle cumplimiento al objetivo principal de este trabajo se requiere comprender el algoritmo criptográfico GOST asimilando conceptos criptográficos relacionados con las técnicas empleadas en él, diseñar e implementar el algoritmo empleando técnicas de programación paralela y validar la solución paralela implementada.

2. DESARROLLO DEL ARTÍCULO

2.1. Descripción del algoritmo GOST

La criptografía se puede clasificar en dos grandes grupos, dependiendo del tipo de llave empleada: criptografía simétrica y asimétrica. La criptografía simétrica se refiere al conjunto de métodos que permiten tener comunicación segura siempre y cuando, anteriormente, hayan intercambiado la llave. La simetría implica que se emplea la misma llave k para cifrar como para descifrar [1, 4].

El GOST es un algoritmo simétrico de cifrado por bloques que emplea una llave de 256 bits, el cual contiene una red Feistel, con 32 iteraciones. Cada iteración contiene una llave de adición módulo de 2^{32} , un conjunto de 8 S-box de 4 bits, y una rotación simple de 11 posiciones. Una particularidad del GOST es que sus S-Cajas pueden ser secretas y pueden ser usadas continuamente como una llave secundaria, la cual es común a cada aplicación; de esta forma el tamaño de la llave se extendería a un total de 610 bits [2, 3].

En el estándar [3] se especifican 4 modos en los que el criptosistema puede trabajar, en este trabajo se emplea el modo de sustitución simple. La Fig. 1 explica el funcionamiento del algoritmo, la que se detalla a continuación:

El modo empleado plantea dividir el texto plano a cifrar en bloques de 64 bits. Cada bloque es copiado en los registros R_1 y R_2. La llave K de 256 bits es dividida en 8 fragmentos de 32 bits y es almacena en el KSU (unidad de almacenamiento de llaves). El cifrado consiste en 32 iteraciones. La primera iteración se caracteriza por el valor inicial de registro R_1 que es adicionado al primer fragmento de la llave. El resultado de esta operación es divido en 8 pedazos de 4 bits y enviado a cada una de las S-Cajas (el bloque S). Luego se hace una rotación de 11 bits a la izquierda. El contenido de es R adicionado al contenido de R_2 y almacenado en R_1 y el valor anterior de R_1 es almacenado en R_2 [3].

Las siguientes iteraciones son similares a la primera, la diferencia radica en el orden en el cual son usadas las llaves. En la segunda iteración, se usa la llave parcial K_1 del KSU. A las iteraciones 3, 4, 5, 6, 7, 8, se les aplica las llaves parciales $K_2, K_3, K_4, K_5, K_6, K_7$ respectivamente. Las iteraciones desde la 9 a la 16 y desde la 17 a la 24 usan las mismas llaves parciales. Las iteraciones desde la 25 a la 32, aplican las llaves parciales en orden inverso, o sea, la iteración 25 usa la llave K_7, la iteración 26 usa la llave K_6 y así sucesivamente [3].

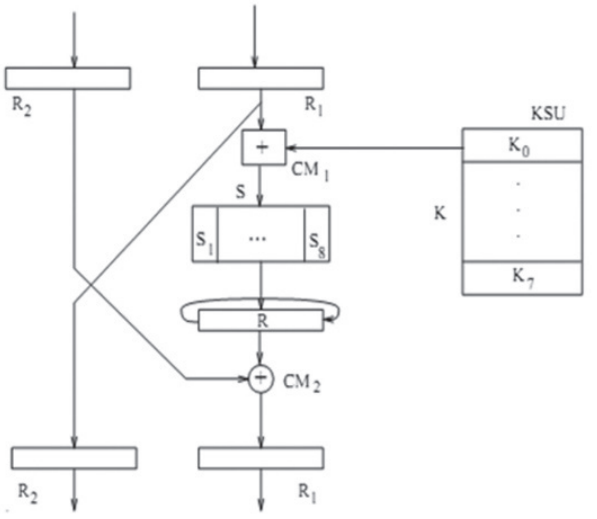

Fig. 1. Funcionamiento del algoritmo GOST

2.2. Análisis y diseño paralelo del algoritmo

La sección anterior caracterizó el funcionamiento del algoritmo GOST. En esta, se realiza un análisis del código implementado con el objetivo de identificar la región de código caliente o región a paralelizar. Se cuenta con una implementación en C/C++ que sigue el modelo mostrado en la Fig. 1.

Se realizan varias ejecuciones del algoritmo divido por secciones para determinar la región de mayor procesamiento.

La Fig. 2 muestra los tiempos obtenidos para diferentes tamaños de ficheros en una arquitectura i7 920. Las pruebas demuestran que el tiempo de ejecución depende del tamaño de los datos cifrar/descifrar.

Las secciones analizadas son: Creación de las S-Cajas y Proceso de Cifre/Descifre. El cifrado y descifrado de un fichero de 750 MB se demora 24 segundos, lo que representa un 70 % del tiempo total del algoritmo.

Dentro del proceso de cifre y descifre se encuentran las 32 iteraciones que se realizan a cada bloque de 64 bits que a su vez contiene la rotación de las 11 posiciones, por lo cual se decide como región a paralelizar el proceso de cifre/descifre de los datos.

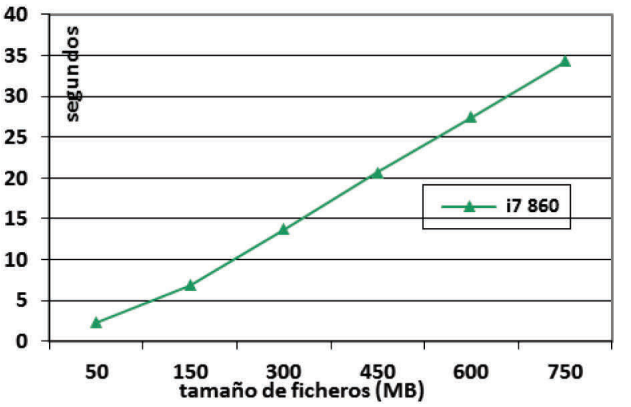

Fig. 2. Tiempo de ejecución del algoritmo con ficheros de diferentes tamaños

El diseño que a continuación se propone hace uso de la metodología de Lan Foster [5]. Esta metodología permite que el programador se concentre primero en la concurrencia y la escalabilidad y después se enfoque en los otros aspectos como son la localidad y el rendimiento. La metodología se divide en 4 etapas. La primera es el particionado, que plantea dividir el problema en tareas más pequeñas. En esta esta etapa suele ignorarse el número de procesadores disponibles en la arquitectura. Existen 2 formas de particionar un algoritmo, una es la descomposición de dominio y la otra es descomposición funcional. La descomposición de dominio implica dividir los datos en tantas partes como sea posible y la descomposición funcional platea dividir el algoritmo en funciones que puedan realizarse concurrentemente. La segunda etapa es la comunicación, la cual se enfoca en el flujo de información y coordinación entre las tareas que son creadas durante la etapa del particionado. La tercera etapa es la de agrupación, en la cual la tarea y la estructura de la comunicación son divididas primeramente en dos etapas de diseño y evaluadas respecto de los requerimientos de ejecución y el costo de implementación. Si es necesario, las tareas son combinadas en tareas más grandes para mejorar el rendimiento o reducir los costos del desarrollo. La última etapa es el mapeo, donde cada tarea es asignada a un procesador de forma tal que se maximice el uso de la mayoría de los procesadores y se minimice el costo de las comunicaciones. El mapeo puede ser específicamente estático o determinado en tiempo de ejecución, buscando el balanceo de carga de los algoritmos [5]. Por tanto se propone:

1- Emplear particionado de dominio, pues no existen funciones que puedan realizarse paralelamente, debido a que el cifrado/descifrado de un bloque es intrínsecamente secuencial. Se necesita crear primero las S-Cajas, luego dividir el bloque a la mitad y cifrarlo con cada porción de la llave correspondiente. La Fig. 1 explica el funcionamiento del algoritmo para un bloque de 64 bits, por lo cual se decide dividir el texto a cifrar/descifrar en bloques de 64 bits como plantea el GOST.

2- Cada bloque de 64 bits puede ser cifrado/descifrado sin depender del siguiente bloque, por lo que la fase de comunicación queda exenta.

3- Realizar grupos en dependencia de la cantidad de hilos que contenga la arquitectura donde se ejecute el algoritmo.

4- Asignar a los grupos creados los hilos de ejecución de forma dinámica, buscando que todos los hilos tengan la misma cantidad de trabajo.

El diseño realizado es aplicado a dos implementaciones empleando el paradigma de memoria compartida, los cuales son: OpenMP y CUDA.

OpenMP es una API para la programación de aplicaciones paralelas. Está conformado por tres elementos principales: directivas de compilación, librerías de rutinas en tiempo de ejecución y variables de entorno. Se basa en el modelo Fork-Join, donde la tarea es dividida entre el hilo máster y un número de hilos esclavos. Los hilos se ejecutan al mismo tiempo y se distribuyen entre los diferentes procesadores [6-8].

CUDA es un lenguaje orientado a C++ propuesto por NVIDIA para la programación y manejo de operaciones e instrucciones sobre una tarjeta gráfica (GPU). Las GPUs, a parir de la serie G8X de NVIDIA, GeForce, Quadro y la línea Tesla son compatibles con CUDA. La arquitectura de CUDA se descompone de manera jerárquica en varias capas: driver del hardware, una API y su runtime, y dos librerías matemáticas de alto nivel de uso común como son CUFFT y CUBLAS. La API es una extensión del lenguaje de programación C y aporta un compilador que redirige la parte que no se ejecuta en la GPU al compilador por defecto del sistema [9-11].

La diferencia fundamental entre los paradigmas descritos se basa en el dispositivo que pueden explotar. OpenMP solo puede emplear la CPU; CUDA es para tarjetas gráficas, pero solo de la compañía NVIDIA [9]. Cada uno de estos añade porciones de códigos a la implementación original. OpenMP es el menos complejo teniendo en cuenta que solo se le adiciona a las secciones a paralelizar los *pragmas*. Los pragmas son sentencias especiales que le indican al compilar que el siguiente código se ejecutará en paralelo [8]. Las tarjetas gráficas cuentan con su propia memoria, por lo que CUDA requiere enviar los datos hacia la GPU, calcular el resultado y enviar el resultado a la memoria RAM [11]. Todas estas características añaden un tiempo adicional; el objetivo fundamental es identificar el paradigma que logre disminuir más el tiempo de ejecución.

3. ANÁLISIS DE RESULTADOS

El acápite presenta la validación de la solución propuesta. Para ello se hace uso de las métricas para evaluar los algoritmos paralelos y se desarrolla una serie de pruebas comparativas. El objetivo de esta sección es comprobar que el tiempo de ejecución del algoritmo implementado disminuye respecto a su versión secuencial e identificar cuál es la técnica de programación paralela que logra reducir más el tiempo logrado con el algoritmo secuencial**.**

3.1. Aceleración

El primer criterio tomado en cuenta en la ejecución de un programa paralelo es analizar la aceleración (Speed Up) obtenida. La aceleración es una medida que indica cuántas veces es más rápido el programa paralelo comparado con el programa secuencial.

La fórmula de la aceleración es la que se expone a continuación [12]:

$$S = \frac{Ts}{Tp}$$

Donde Ts es el tiempo de ejecución secuencial y Tp es el tiempo de ejecución paralelo.

3.2. Eficiencia

La eficiencia paralela es una medida que denota qué tan bien se han utilizado los procesadores, o sea, es la fracción de tiempo que los procesadores emplean para realizar las tareas asignadas. No depende del tamaño del problema [12].

Tabla 1. Características del hardware empleado

Hardware	Tipo	Memoria						
		Elemento de Cómputo	Reloj (MHZ)	Tipo	Cantidad (MB)	Reloj (MHz)	Ancho de Bus (bits)	Ancho de banda (GB/s)
Intel Core 2 Duo E7300	CPU	2	2600	DDR2	2048	800	64	12.4
Intel Core 2 Quad Q9300	CPU	4	2500	DDR2	4096	800	64	12.4
Intel Core i7 920	CPU	8	2670	DDR3	6144	1066	64	25.5
nVidia Geforce GTX 260	GPU	216	1242	DDR3	896	1998	448	111.9
nVidia Geforce GTX 550 Ti	GPU	192	1800	DDR5	1024	4104	192	98.5
nVidia Geforce GTX 660	GPU	960	980	DDR5	2048	3004	192	144

Tabla 2. Tiempos obtenidos (segundos) con las implementaciones Secuencial (Sec), OpenMP y CUDA para cada una de las Arquitecturas (Arq) de pruebas con ficheros de distintos tamaños

	50 MB	150 MB	300 MB	450 MB	600 MB	750 MB
Arq 1 Sec	2.50	7.47	14.96	22.43	29.85	37.39
Arq 2 Sec	2.65	7.94	15.88	23.83	31.78	39.70
Arq 3 Sec	2.29	6.88	13.74	20.66	27.44	34.32
Arq 1 OpenMP	1.25	3.76	7.50	11.23	14.96	18.72
Arq 2 OpenMP	0.66	1.99	3.99	6.00	7.69	9.97
Arq 3 OpenMP	0.45	1.33	2.65	4.11	5.31	6.66
Arq 1 CUDA	0.08	0.24	0.48	0.73	0.93	1.09
Arq 2 CUDA	0.28	0.83	1.47	1.62	1.76	1.95
Arq 3 CUDA	0.07	0.22	0.42	0.65	0.84	0.94

Se puede definir la Eficiencia como:

$$E = \frac{Ts}{P \times Tp}$$

Dónde Ts es el tiempo de ejecución secuencial, Tp es el tiempo de ejecución paralelo y P número de procesadores empleados.

El valor máximo teórico de esta ecuación es 1.

3.3. Escenarios de Prueba

La validación de los resultados se realiza teniendo en cuenta las métricas descritas y varios escenarios de pruebas, los cuales se detallan a continuación:

Arquitectura 1: Intel Core 2 Duo y una tarjeta de video nVidia Geforce GTX 550 Ti.

Arquitectura 2: Intel Core 2 Quad y una tarjeta de video nVidia Geforce GTX 260.

Arquitectura 3: Intel Core i7 920 y una tarjeta de video nVidia Geforce GTX 660.

Las características más significativas de las arquitecturas descritas se especifican en la tabla 1.

Las pruebas se realizaron con cinco ficheros con tamaños 50, 150, 300, 450, 600 y 750 MB respectivamente. El tiempo de ejecución que se grafica en cada figura es el promedio de un total de 15 iteraciones del algoritmo en cada arquitectura descrita. El sistema operativo donde se ejecutaron las pruebas fue Centos 6.4. Los resultados obtenidos con las implementaciones paralelas son los mismos que se logran con la implementación secuencial.

El siguiente experimento se realiza con las tres arquitecturas de prueba. Los tiempos que se muestran son los obtenidos con las tres implementaciones.

La tabla 2 muestra que los mejores tiempos de ejecución se logran con la implementación realizada en CUDA.

A continuación se realiza un análisis teniendo en cuenta el *speed-up* y la eficiencia alcanzada para cada arquitectura de prueba.

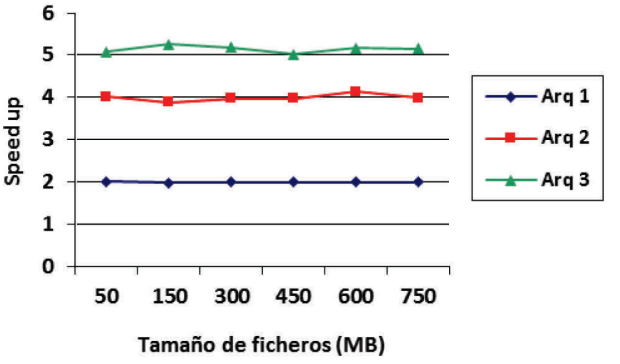

Fig. 3. *Speed up* obtenido con la implementación OpenMP para las tres arquitecturas de prueba

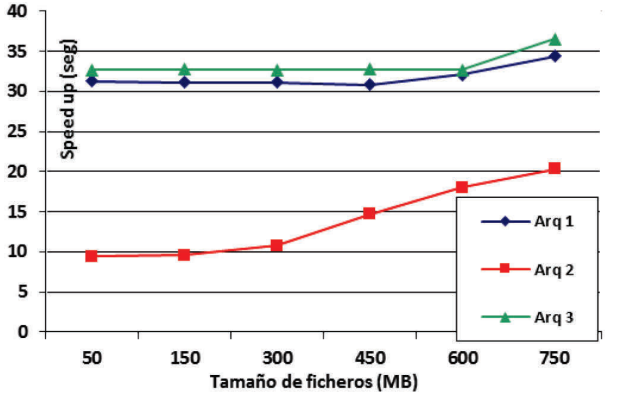

Fig. 4. *Speed up* obtenido con la implementación CUDA para las tres arquitecturas de prueba

La Fig. 3 muestra que la ganancia de velocidad se mantiene constante respecto a la obtenida de forma secuencial, aunque la mayor ganancia se logra con la arquitectura 3.

La Fig. 4 muestra para la arquitectura 1 y 2 una ganancia de velocidad constante respecto a la obtenida de forma secuencial con cada uno de los ficheros, aunque para la arquitectura 1, a partir de ficheros con 450 MB el *speed up* comienza a crecer en 2 unidades más de lo esperado. La arquitectura 3 tiene un comportamiento similar al descrito para la arquitectura 2, pero para ficheros mayores de 450 MB. El comportamiento del algoritmo para la arquitectura 2 sí crece de forma exponencial respecto al obtenido con el algoritmo secuencial. La arquitectura que logra obtener el mejor *speed up* es la 3.

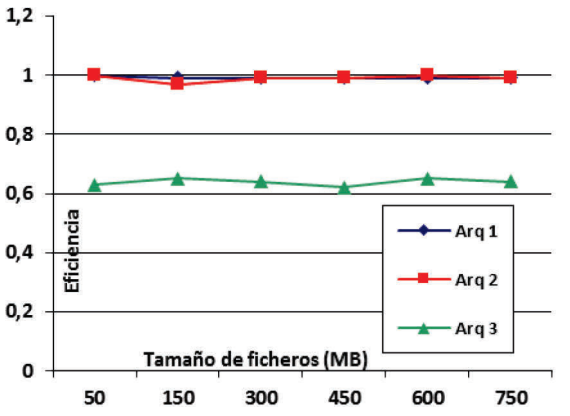

Fig. 5. Eficiencia obtenida con la implementación OpenMP en cada arquitectura de prueba

La Fig. 5 muestra que los procesadores de las arquitecturas 1 y 2 se emplean eficientemente aproximadamente un 99 %, en cambio en la arquitectura 3 solo se logra emplear eficientemente un 64 %.

4. CONCLUSIONES

El artículo presenta dos implementaciones paralelas empleando OpenMP y CUDA. Ambas implementaciones logran reducir el tiempo de la ejecución secuencial, pero los mejores tiempos se logran con CUDA, específicamente con la tarjeta 660 GTX de la arquitectura de prueba 3. El proceso de cifre y descifre para esta arquitectura se realiza en 0.94 segundos para un fichero de 750 MB, lo que significa que el algoritmo paralelo con CUDA es aproximadamente 37 veces más rápido que el tiempo secuencial. Se recomienda implementar el algoritmo empleando OpenCL y comparar los tiempos obtenidos para CPU con OpenMP y para GPU con CUDA.

REFERENCIAS

[1] H. C. Van Tilborg & S. Jajodia, *Encyclopedia of cryptography and security.* Springer Science & Business Media, 2011. 1416p. ISBN 978-1-4419-5907-2

[2] N. T. Courtois, "Security Evaluation of GOST 28147-89 In View Of International Standardisation," *Cryptologia*, Vol. 36, no. 1, 2012, pp. 2-13. DOI:10.1080/01611194.2011.632807

[3] J. Pieprzyk & L. Tombak, "*Soviet Encryption Algorithm*", University of Wollongong, Department of Computing Science, 1994.

[4] N. Ferguson, B. Schneier, T. Kohno, "*Cryptography engineering: design principles and practical applications",* John Wiley & Sons, 2011, 384p. ISBN: 978-0-470-47424-2

[5] I. Foster, "Designing and building parallel programs," Addison Wesley Publishing Company, 1995, 430p. ISBN: 978-0201575941

[6] OpenMP, "The OpenMP API specification for parallel programming," 2010 URL http://openmp.org.

[7] OpenMP, A. R. Board, "OpenMP Application Program Interface 3.0", 2008. URL: http://www.openmp.org/mp-documents/spec30.pdf

[8] R. Chandra, "Parallel programming in OpenMP" en *Morgan Kaufmann Publishers Inc*, 2001, 248p. ISBN: 978-1558606715

[9] N. Wilt, "CUDA Handbook: A Comprehensive Guide to GPU Programming", 1st ed. Addison-Wesley Professional, 2013, 528p. ISBN: 978-0321809469

[10] S. Cook, "CUDA programming: a developer's guide to parallel computing with GPUs", Newnes, 2012, 576p. ISBN: 978-0124159334

[11] J. Sanders & E. Kandrot, "CUDA by example: an introduction to general-purpose GPU programming*",* Addison-Wesley Professional, 2010, 312p. ISBN: 978-0131387683

[12] A. Grama, A. Gupta, G. Karyspis, V. Kumar, "Introduction to Parallel Computing", 2nd ed. Addison Wesley, 2003, 656p. ISBN: 978-0201648652