



UNIVERSIDAD DE CÓRDOBA

Programa de doctorado:

Computación Avanzada, Energía y Plasmas

Optimización de Algoritmos Científicos en
Sistemas Heterogéneos y Aceleradores
para Computación de Altas Prestaciones

Optimization of Scientific Algorithms in
Heterogeneous Systems and Accelerators
for High Performance Computing

Directores:

Dr. Juan Gómez Luna

Dr. Rafael Medina Carnicer

Autor:

D. Antonio Fuentes Alventosa

Fecha de depósito:

Enero de 2023

TITULO: *Optimización de Algoritmos Científicos en Sistemas Heterogéneos y Aceleradores para Computación de Altas Prestaciones*

AUTOR: *Antonio Fuentes Alventosa*

© Edita: UCOPress. 2023
Campus de Rabanales
Ctra. Nacional IV, Km. 396 A
14071 Córdoba

<https://www.uco.es/ucopress/index.php/es/>
ucopress@uco.es

**TÍTULO DE LA TESIS:**

Optimización de Algoritmos Científicos en Sistemas Heterogéneos y Aceleradores para Computación de Altas Prestaciones

DOCTORANDO/A:

Antonio Fuentes Alventosa

INFORME RAZONADO DEL/DE LOS DIRECTOR/ES DE LA TESIS

(se hará mención a la evolución y desarrollo de la tesis, así como a trabajos y publicaciones derivados de la misma)

Consideramos que el doctorando ha alcanzado los objetivos de la tesis doctoral con la paralelización de tres importantes algoritmos científicos utilizando procesadores gráficos (GPU).

Productos del trabajo son tres artículos en revistas indexadas JCR, que le permiten presentar la tesis doctoral por compendio de artículos:

- Fuentes-Alventosa, A., Gómez-Luna, J., González-Linares, J. M., Guil, N., & Medina-Carnicer, R. (2022). CAVLCU: an efficient GPU-based implementation of CAVLC. *The Journal of Supercomputing*, 78(6), 7556-7590.
 - Journal Impact Factor (2021): 2.557 (Q2).
 - Category: Computer Science, Theory & Methods.

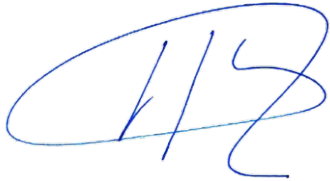
- Fuentes-Alventosa, A., Gómez-Luna, J., & Medina-Carnicer, R. (2022). GUD-Canny: A real-time GPU-based unsupervised and distributed Canny edge detector. *Journal of Real-Time Image Processing*, 19(3), 591-605.
 - Journal Impact Factor (2021): 2.293 (Q3).
 - Category: Computer Science, Artificial Intelligence.

- Fuentes-Alventosa, A., Gómez-Luna, J., & Medina-Carnicer, R. (2022). GVLE: a highly optimized GPU-based implementation of variable-length encoding. *The Journal of Supercomputing*, 1-28.
 - Journal Impact Factor (2021): 2.557 (Q2).
 - Category: Computer Science, Theory & Methods.

Por todo ello, se autoriza la presentación de la tesis doctoral.

Córdoba, 11 de enero de 2023

Firma de los directores

A handwritten signature in blue ink, consisting of a large, stylized 'J' followed by 'G' and 'L'.

Fdo.: Juan Gómez Luna

A handwritten signature in black ink, starting with a large 'R' followed by 'Medina' and 'Carnicer'.

Fdo.: Rafael Medina Carnicer

Resumen

Actualmente, la computación de propósito general en GPU es uno de los pilares básicos de la computación de alto rendimiento. Aunque existen cientos de aplicaciones aceleradas en GPU, aún hay algoritmos científicos poco estudiados. Por ello, la motivación de esta tesis ha sido investigar la posibilidad de acelerar significativamente en GPU un conjunto de algoritmos pertenecientes a este grupo.

En primer lugar, se ha obtenido una implementación optimizada del algoritmo de compresión de vídeo e imagen CAVLC (Context-Adaptive Variable Length Encoding), que es el método entrópico más usado en el estándar de codificación de vídeo H.264. La aceleración respecto a la mejor implementación anterior está entre 2.5x y 5.4x. Esta solución puede aprovecharse como el componente entrópico de codificadores H.264 software, y utilizarse en sistemas de compresión de vídeo e imagen en formatos distintos a H.264, como imágenes médicas.

En segundo lugar, se ha desarrollado GUD-Canny, un detector de bordes de Canny no supervisado y distribuido. El sistema resuelve las principales limitaciones de las implementaciones del algoritmo de Canny, que son el cuello de botella causado por el proceso de histéresis y el uso de umbrales de histéresis fijos. Dada una imagen, esta se divide en un conjunto de sub-imágenes, y, para cada una de ellas, se calcula de forma no supervisada un par de umbrales de histéresis utilizando el método de Medina-Carnicer. El detector satisface el requisito de tiempo real, al ser 0.35 ms el tiempo promedio en detectar los bordes de una imagen 512x512.

En tercer lugar, se ha realizado una implementación optimizada del método de compresión de datos VLE (Variable-Length Encoding), que es 2.6x más rápida en promedio que la mejor implementación anterior. Además, esta solución incluye un nuevo método scan inter-bloque, que se puede usar para acelerar la propia operación scan y otros algoritmos, como el de compactación. En el caso de la operación scan, se logra una aceleración de 1.62x si se usa el método propuesto en lugar del utilizado en la mejor implementación anterior de VLE.

Esta tesis doctoral concluye con un capítulo sobre futuros trabajos de investigación que se pueden plantear a partir de sus contribuciones.

A mis seres queridos, del cielo y de la tierra

Agradecimientos

En primer lugar, a mis directores, Dr. Juan Gómez Luna y Dr. Rafael Medina Carnicer, por sus valiosas orientaciones en la dirección de esta tesis doctoral.

A mis hermanos, José María y Javier, de los que no me puedo sentir más orgulloso, por la fuerza y el entusiasmo que me han transmitido, y a mis sobrinos, Sara, María y José María, inmejorables reflejos de sus padres, por la alegría que me han regalado.

A mi compañera de viaje, María del Mar, por su apoyo sin límites. No tengo palabras para agradecer su comprensión, su generosidad y su optimismo.

Por último, a mis maravillosos padres, Antonio y Gema, por no sólo darme la vida, sino por iluminar constantemente su trayecto y ser mis mejores referentes.

Índice

1. Introducción	13
1.1. CUDA	14
1.2. Antecedentes	16
1.3. Objetivos	17
2. Contribuciones	19
2.1. Primera contribución: "CAVLCU: an efficient GPU-based implementation of CAVLC"	19
2.2. Segunda contribución: "GUD-Canny: a real-time GPU-based unsupervised and distributed Canny edge detector"	57
2.3. Tercera contribución: "GVLE: a Highly Optimized GPU-Based Implementation of Variable-Length Encoding"	75
3. Conclusiones y futuros trabajos	105
3.1. Conclusiones	105
3.2. Futuros trabajos	106
Referencias	107

1. Introducción

Hoy en día, la computación de propósito general en unidades de procesamiento de gráficos, denominada abreviadamente *GPGPU* (General Purpose Computing on Graphics Processing Units), es una de las tendencias principales en la computación de alto rendimiento [1]. En los años 2000, la creciente demanda del mercado de videojuegos causó un incremento muy significativo de la capacidad de computación de las GPUs, en contraste con la más lenta evolución de las CPUs [2]. Este hecho ofreció la gran oportunidad de acelerar la resolución de problemas de propósito general mediante un procesamiento repartido entre CPU y GPU (en lugar de realizarlo exclusivamente la CPU) [2]. En 2007, para facilitar este sistema heterogéneo de computación, NVIDIA lanzó la arquitectura de cálculo paralelo CUDA (Compute Unified Device Architecture) [3], que, desde entonces, viene incorporada en sus tarjetas gráficas. CUDA proporciona un conjunto de herramientas de desarrollo y extensiones a lenguajes de programación de alto nivel, como C y C++, que facilitan enormemente la labor del programador [4, 5].

Desde hace una década y media, se están acelerando cientos de aplicaciones en GPU en los más diversos campos [6], como el aprendizaje automático [7, 8], el procesamiento de vídeo e imagen [9, 10] y la minería de datos [11, 12]. Sin embargo, aún existen algoritmos científicos poco explorados, como los métodos de compresión de datos CAVLC (Context-based Adaptive Variable Length Coding) [13, 14], CABAC (Context-based Adaptive Binary Arithmetic Coding) [15] y VLE (Variable-Length Encoding) [16, 17], y diferentes métodos de procesamiento de vídeo e imagen, como la detección de bordes de Canny no supervisada [18], el cálculo del descriptor de imágenes RCD (Region Covariance Descriptor) [19], la identificación de texturas a gran escala en tiempo real [20] y el algoritmo de eliminación de ruido de vídeo VBM3D (Video Block-Matching and 3-D Filtering) [21]. Por ello, la motivación de esta tesis ha sido investigar la posibilidad de acelerar significativamente en GPU un conjunto de algoritmos científicos pertenecientes a este grupo.

El resto de este capítulo introductorio se divide en las siguientes secciones. La sección 1.1 proporciona un breve resumen de CUDA para facilitar la comprensión de las optimizaciones desarrolladas en esta tesis, la sección 1.2 presenta sus antecedentes, y la 1.3, sus objetivos.

1.1. CUDA

CUDA [3] es una plataforma de computación paralela de propósito general que aprovecha el motor de cálculo paralelo de sus GPUs para la resolución de problemas computacionalmente complejos de forma mucho más eficiente que usando sólo una CPU [4].

Los objetivos de diseño de CUDA fueron los siguientes [4]:

- Proporcionar un conjunto pequeño de extensiones a lenguajes de programación estándar, como C, C++, Fortran, Python y MATLAB, con objeto de facilitar enormemente el desarrollo de algoritmos paralelos [3]. De esta forma, se eliminó la necesidad de tener que usar lenguajes de programación específicos para gráficos (como OpenGL o Cg) para la resolución de problemas de propósito general, lo cual es mucho más complejo. En este trabajo, se ha utilizado CUDA C++ [4, 5].
- Soportar computación heterogénea, consistente en que las aplicaciones ejecuten sus partes secuenciales en la CPU y las paralelas en la GPU. La CPU y la GPU (denominadas *host* y *device*, respectivamente) se modelan como dispositivos separados, con sus propios espacios de memoria.

CUDA permite definir funciones, denominadas *kernels*, que son ejecutadas por muchos hilos en paralelo en la GPU [4]. Los hilos se organizan en *bloques*, que pueden tener entre una y tres dimensiones. Un kernel es ejecutado por un conjunto de bloques idénticos, denominado *grid*, cuyo número de dimensiones también puede ser hasta tres.

La arquitectura de una GPU de CUDA se compone de un conjunto de *multiprocesadores de streaming* [4]. Cuando un programa que está siendo ejecutado por la CPU llama a un kernel, los bloques del correspondiente grid se distribuyen a los multiprocesadores con capacidad de ejecución disponibles. Los hilos de cada bloque se ejecutan concurrentemente en un único multiprocesador, y, a su vez, cada multiprocesador puede ejecutar concurrentemente muchos bloques. Conforme va finalizando la ejecución de los bloques, otros nuevos se van lanzando en los multiprocesadores vacantes. Para que un kernel escale con el número de multiprocesadores de cualquier

tarjeta gráfica, los bloques se deben poder ejecutar independientemente, es decir, en cualquier orden y en paralelo o en serie.

Un multiprocesador ejecuta los bloques en grupos de 32 hilos consecutivos, denominados *warps* [4]. Cada bloque, tras ser asignado a un multiprocesador, es dividido en warps, y cada warp es gestionado por un *planificador de warps*. Los hilos de un warp comienzan en la misma dirección de programa, pero cada uno tiene sus propios contador de programa y registro de estado, por lo que diferentes conjuntos de hilos de un mismo warp pueden seguir caminos de ejecución independientes. Aunque, a efectos de la corrección del código, no es necesario tener en cuenta esta característica de la arquitectura de CUDA, se pueden conseguir mejoras significativas en el rendimiento disminuyendo al máximo las divergencias de ejecución de los warps.

Los hilos de CUDA pueden acceder a los siguientes espacios de memoria [4] durante su ejecución:

- *Memoria privada* de cada hilo, constituida por *registros* y *memoria local*, cuyo tiempo de vida coincide con el del hilo.
- *Memoria compartida*, de baja latencia y visible a todos los hilos de un bloque, cuyo tiempo de vida es el del bloque.
- Un conjunto de memorias usadas para la compartición de datos entre todos los hilos de un grid, que son una *memoria global* de lectura/escritura y otras dos de lectura: la *memoria de constantes*, usada para almacenar valores no modificables, y la *memoria de texturas*, optimizada para accesos con localidad espacial 2D. Los contenidos de estas memorias son persistentes entre llamadas a kernels de una misma aplicación.

La memoria global es la más abundante de todas [5]. Por otro lado, las memorias global, local y de texturas tienen la latencia más alta, seguidas de la memoria de constantes, la memoria compartida y el espacio de registros [5].

Una técnica de optimización muy importante es el *acceso coalescente* a la memoria global [5]. Cuando un warp realiza una operación en memoria global, los accesos a memoria de sus hilos se unen en una o más transacciones de memoria, según el tamaño de las palabras accedidas y la distribución de las direcciones de memoria. Cuanto más

dispersos estén los accesos, más transacciones son necesarias y, por tanto, más se reduce el rendimiento.

1.2. Antecedentes

Aunque actualmente hay cientos de aplicaciones aceleradas en GPU en los más diversos campos [6], aún existen algoritmos científicos poco explorados, como los métodos de compresión de datos CAVLC [13, 14] y VLE [16, 17].

En el caso de CAVLC, la única implementación en GPU anterior a esta tesis es la propuesta por Su et al. [13, 14], desarrollada en CUDA. Satisface el requisito de tiempo real para el formato HDTV 720p, y su rendimiento es entre 6.29 y 11.17 veces mayor que el de los sistemas DSP y multinúcleo publicados hasta ese momento. Aunque se trata de una solución eficiente, existen diferentes factores que limitan su velocidad. El principal de ellos es que, al estar implementada con varios kernels, existe una penalización importante en la eficiencia causada por:

- Los accesos a memoria global de elevada latencia que se han de realizar para transmitir resultados intermedios entre kernels.
- Las costosas llamadas y finalizaciones de los distintos kernels.

En relación a las implementaciones de VLE en GPU, los antecedentes de este trabajo son el algoritmo PAVLE de Balevic [16] y la solución de Rahmani et al. [17]. La evaluación experimental mostró que la aceleración respecto a la implementación en CPU fue 35x, en el primer caso, y 22x, en el segundo. El principal factor que limita la velocidad de estas dos soluciones es el mismo que el de la implementación de CAVLC de Su et al. [13, 14].

Por último, indicar que, antes de esta tesis, se obtuvo el algoritmo CUVLE [22], una implementación eficiente en GPU de VLE. La evaluación experimental mostró que CUVLE es más de dos veces más rápido que PAVLE [16].

1.3. Objetivos

El objetivo general de esta tesis ha sido investigar las posibilidades de acelerar un conjunto de algoritmos científicos en entornos heterogéneos basados en GPU, con la finalidad de realizar contribuciones en cada uno de los siguientes objetivos específicos:

1. Estudiar la posibilidad de optimizar el método de codificación entrópica CAVLC del estándar de compresión de vídeo H.264.
2. Profundizar en la investigación sobre la optimización de VLE en arquitecturas paralelas de última generación.
3. Explorar la posibilidad de optimizar aplicaciones de vídeo e imagen en arquitecturas paralelas a determinar durante el transcurso de la investigación.

2. Contribuciones

2.1. Primera contribución: "CAVLCU: an efficient GPU-based implementation of CAVLC"

En este trabajo [23] se presenta CAVLCU, una implementación eficiente de CAVLC en GPU, que es útil en los siguientes escenarios:

- Se puede aprovechar como el componente de codificación entrópica en codificadores H.264 software, una alternativa adecuada a los codificadores H.264 hardware de las tarjetas gráficas [24] cuando estos últimos carecen de cierta funcionalidad necesaria, como encriptación de datos [25, 26, 27, 28] y ocultación de información [29, 30, 31, 32].
- Se puede utilizar en una amplia variedad de sistemas de compresión basados en GPU para codificar imágenes y vídeos en formatos diferentes a H.264, como imágenes médicas [33, 34, 35]. Esto no es posible con las implementaciones hardware de CAVLC, al no ser componentes separables de los codificadores H.264 hardware en los que están integradas.

CAVLCU se basa en cuatro ideas clave:

- Está compuesto de un solo kernel para evitar los accesos de elevada latencia a memoria global para transmitir resultados intermedios entre kernels, así como los costosos lanzamientos y terminaciones de estos últimos.
- Se aplica un mecanismo eficiente de sincronización entre bloques de hilos que procesan regiones adyacentes del fotograma (en las dimensiones horizontal y vertical) para compartir resultados en memoria global.
- Se explota completamente el ancho de banda de memoria global disponible mediante accesos vectorizados que almacenan directamente los coeficientes transformados cuantificados en registros.
- Se realiza la ordenación en zigzag de los bloques de coeficientes directamente en el espacio de registros con un alto grado de paralelismo a nivel de instrucción.

La evaluación experimental mostró que CAVLCU es entre 2.5x y 5.4x más rápido que la mejor implementación anterior en GPU de CAVLC [13].

Esta contribución se corresponde, como resultado, con los objetivos 1 y 3.



CAVLCU: an efficient GPU-based implementation of CAVLC

Antonio Fuentes-Alventosa¹ · Juan Gómez-Luna² ·
José Maria González-Linares³ · Nicolás Guil³ · R. Medina-Carnicer¹

Accepted: 27 October 2021 / Published online: 29 November 2021
© The Author(s) 2021

Abstract

CAVLC (Context-Adaptive Variable Length Coding) is a high-performance entropy method for video and image compression. It is the most commonly used entropy method in the video standard H.264. In recent years, several hardware accelerators for CAVLC have been designed. In contrast, high-performance software implementations of CAVLC (e.g., GPU-based) are scarce. A high-performance GPU-based implementation of CAVLC is desirable in several scenarios. On the one hand, it can be exploited as the entropy component in GPU-based H.264 encoders, which are a very suitable solution when GPU built-in H.264 hardware encoders lack certain necessary functionality, such as data encryption and information hiding. On the other hand, a GPU-based implementation of CAVLC can be reused in a wide variety of GPU-based compression systems for encoding images and videos in formats other than H.264, such as medical images. This is not possible with hardware implementations of CAVLC, as they are non-separable components of hardware H.264 encoders. In this paper, we present CAVLCU, an efficient implementation of CAVLC on GPU, which is based on four key ideas. First, we use only one kernel to avoid the long latency global memory accesses required to transmit intermediate results among different kernels, and the costly launches and terminations of additional kernels. Second, we apply an efficient synchronization mechanism for thread-blocks (In this paper, to prevent confusion, a block of pixels of a frame will be referred to as simply *block* and a GPU thread block as *thread-block*.) that process adjacent frame regions (in horizontal and vertical dimensions) to share results in global memory space. Third, we exploit fully the available global memory bandwidth by using vectorized loads to move directly the quantized transform coefficients to registers. Fourth, we use register tiling to implement the zigzag sorting, thus obtaining high instruction-level parallelism. An exhaustive experimental evaluation showed that our approach is between 2.5× and 5.4× faster than the only state-of-the-art GPU-based implementation of CAVLC.

Keywords CAVLC · GPU · CUDA · H.264 · Parallel implementations · Data compression · Variable-length encoding

Extended author information available on the last page of the article

1 Introduction

In the current digital era, the massive use of multimedia data, such as images and videos, together with the necessity to overcome the restrictions of storage space and communication bandwidth, have given an essential role to data compression.

Generally speaking, data compression can be *lossless* or *lossy*, depending on whether the original content is preserved or not [33]. Lossless compression is used when it is necessary that the original and uncompressed data remain exactly the same, such as executable programs and textual documents. Lossy compression discards some information to increment the amount of data reduction. Image file formats like PNG use only lossless compression, while others like TIFF may use either lossless or lossy methods [2]. *Entropy coding* [33] is a type of lossless compression in which mostly used patterns are assigned with codes of shorter length, whereas rarely used patterns are assigned with codes of longer length.

CAVLC (Context-Adaptive Variable Length Coding) is a high-performance entropy technique for video and image compression [14, 32]. In this method, different sets of variable-length codes are chosen depending on already encoded syntax elements. It is the most commonly used entropy technique in the video standard H.264.

In the last two decades, many designs for CAVLC have been proposed. The majority of these solutions are based on hardware, such as FPGA [4, 6, 12, 28] and ASIC approaches [1, 3, 13, 21]. In contrast, parallel software implementations of CAVLC [5, 31, 38, 39, 43] are currently very scarce.

One of the most successful trends in high-performance computing is general-purpose computation on graphics processing units (*GPGPU*), thanks to programming environments such as CUDA [26] and OpenCL [15]. Efficient implementations of CAVLC on GPU are currently very useful for the following reasons. First, they can be exploited as the entropy component in GPU-based H.264 encoders, which are a very suitable solution when it is necessary to implement functionality not provided by GPU built-in H.264 hardware encoders (e.g., NVENC in NVIDIA graphics cards [27]). In that case, many adaptations of CAVLC proposed in different fields, like data encryption [19, 40–42] and information hiding [16, 17, 45, 46], can be applied. Second, implementations of CAVLC on GPU can be reused and easily adapted in the development of a great variety of GPGPU compression systems for encoding both images and videos in formats other than H.264, like medical images [20, 30, 37]. This is not possible with hardware implementations of CAVLC, as they are non-separable components of hardware H.264 encoders.

In this paper, we present CAVLCU, an optimized implementation of CAVLC on GPU developed in CUDA. As our approach is built using only one CUDA kernel, it avoids the long latency global memory accesses required to transmit intermediate results among different kernels, and the costly launches and terminations of additional kernels. In our algorithm, thread-blocks that process adjacent frame regions (in horizontal and vertical dimensions) share results in global memory space using an efficient synchronization mechanism. Additionally, CAVLCU simplifies the zigzag sorting of the blocks, as each thread, after reading its block

through a vectorized load, sorts it efficiently in the register space through few high throughput operations with high degree of instruction-level parallelism.

Therefore, our main contributions in this work are the following. First, a highly optimized GPU-based approach to CAVLC implemented in CUDA. Second, comparison of our implementation with the only existing state-of-the-art GPGPU implementation [38, 39]. An exhaustive experimental evaluation showed that our solution is between 2.5× and 5.4× faster than the state-of-the-art implementation [38, 39].

The rest of the paper is organized as follows. Sections 2 and 3 give background for CAVLC and the state-of-the-art GPU-based implementation of CAVLC [38, 39], respectively. Section 4 presents CAVLCU. Section 5 shows the experimental evaluation of our algorithm and a comparison to the state-of-the-art solution [38, 39]. Section 6 presents applications of CAVLC. Finally, the main conclusions are stated in Sect. 7.

2 Context-adaptive variable length coding (CAVLC)

CAVLC (Context-Adaptive Variable Length Coding) [14, 32] is a high efficient entropy method for encoding the quantized transform coefficients in video and image compression. In this technique, different sets of variable-length codes are chosen depending on already encoded syntax elements. Since the variable-length codes are designed to match the corresponding conditioned statistics, the entropy coding performance is improved by 5-10% in comparison to prior standards designs (like MPEG, H.261/3) using a single variable-length code.

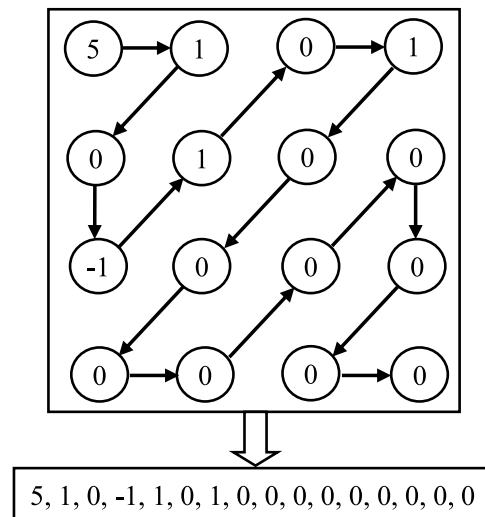
CAVLC is one of the two entropy methods in H.264 [14], the most widely used video coding standard [29]. The alternative is CABAC (Context-Adaptive Binary Arithmetic Coding) [32], a method of arithmetic coding in which the probability models are updated based on previous coding statistics. Compared to CABAC, CAVLC has lower compression efficiency, but higher coding speed and lower complexity. Thus, it is widely used in low-delay, ‘conversational’ applications such as video conferencing, with relatively low computational requirements. Moreover, CAVLC is supported in all H.264 profiles, unlike CABAC which is not supported in baseline and extended profiles.

Next, Subsection 2.1 gives a detailed description of CAVLC algorithm, and Subsection 2.2 presents an example to clarify its operation.

2.1 CAVLC algorithm

CAVLC operates on blocks of 4×4 and 2×2 coefficients. It follows the steps presented in Algorithm 1 for encoding a block [32]. First, as shown in Fig. 1 for a 4×4 block, the coefficients are scanned in zigzag order. The resulting array will be referred to as *zigzag array* in the rest of the paper. Then, CAVLC constructs the output bitstream by concatenating a series of binary variable length codes (VLCs) assigned to the following data elements (*symbols*) of the zigzag array: *CoeffToken*,

Fig. 1 Example of zigzag scan for a 4×4 block



trailing ones, *levels*, *TotalZeros* and *runs*. In the next subsections, we define the referenced symbols and describe how they are encoded.

Algorithm 1: CAVLC block encoding. The operator $\&$ represents binary concatenation

```

zigzag array  $\leftarrow$  scan the coefficients of the block
                    in zigzag order

encoding  $\leftarrow$  VLC of symbol CoeffToken of zigzag array
                    &
                    Bit signs of trailing ones of zigzag array
                    &
                    VLCs of levels of zigzag array
                    &
                    VLC of symbol TotalZeros of zigzag array
                    &
                    VLCs of runs of zigzag array
    
```

2.1.1 CoeffToken

The magnitude of nonzero coefficients tends to be larger at the start of the zigzag array, near the first coefficient, and smaller towards the higher frequencies. In addition, the absolute value of the last nonzero coefficients often equals to 1. The last up to three -1 or +1 coefficients are referred to as *trailing ones (TIs)*, while the remaining nonzero coefficients as *levels*. The symbol *CoeffToken* (coefficient token) represents both the total number of nonzero coefficients (*TotalCoeff*) and the number of trailing ones (*NumTIs*).

The VLC assigned to *CoeffToken* is obtained from a lookup table that, in the case of a 4 × 4 block, is chosen from three VLC tables and one 6-bit fixed length code table, whose contents are specified in Table 9-5 of the H.264 standard [14]. An extract of this latter is shown in Table 1. As it can be seen, the choice of the

Table 1 Extract of the table 9-5 of the H.264 standard

NumT1s	TotalCoeff	$0 \leq nC < 2$	$2 \leq nC < 4$	$4 \leq nC < 8$	$8 \leq nC$
0	0	1	11	1111	0000 11
0	1	0001 01	0010 11	0011 11	0000 00
.....					
.....					
2	5	0000 0010 1	0000 101	0100 1	0100 10
3	5	0000 100	0011 0	1010	0100 11
.....					
.....					
2	16	0000 0000 0000 0101	0000 0000 0001 01	0000 0000 11	1111 10
3	16	0000 0000 0000 1000	0000 0000 0001 00	0000 0000 10	1111 11

lookup table is done in function of a parameter nC , which is calculated from the number of coefficients in the blocks to the left and above of the current block (parameters nA and nB , respectively). This implies that the lookup table selection is context adaptive. Figure 2 illustrates the relationship between a block and its neighbours. The parameter nC is calculated as shown in Table 2, where \gg indicates binary right shift. The availability of each neighbouring block is determined by its existence and its membership in the same slice of the current block.

Fig. 2 Relationship between current block and its top and left neighbours

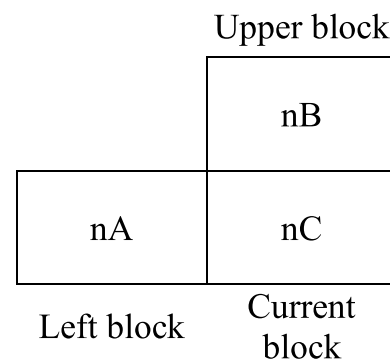


Table 2 Calculation of parameter nC . The operator \gg represents binary right shift

Condition	nC
Left and top blocks are available	$(nA + nB + 1) \gg 1$
Only the left block is available	nA
Only the top block is available	nB
Neither neighbouring block is available	0

2.1.2 Trailing ones

The T1s are encoded in reverse order with their sign bits ('0' for positive and '1' for negative).

2.1.3 Levels

The levels are encoded in reverse order with VLCs composed of a prefix and a possible suffix. The prefix is made up of a string of zero or more bits '0' followed by a stop bit '1'. The length of the suffix (*SuffixLength*) is between 0 and 6 in normal cases. If *SuffixLength* > 0, the last bit of the suffix stores the sign of the level.

Table 3 shows an extract of the seven VLC tables used for levels encoding in the H.264 baseline profile [11], each one corresponding to a different value of

Table 3 VLC Tables of levels

Lev-VLCT[0]		
Level	VLC	Length
+1	1	1
-1	01	2
+2	001	3
-2	0001	4
...
+7	0000000000001	13
-7	00000000000001	14
± 8 to ± 15	000000000000001xxxx	19
≥ ±16	0000000000000001xxxxxxxxxxxxxx	28

Lev-VLCT[1]		
Level	VLC	Length
± 1	1x	2
± 2	01x	3
...
± 5	00001x	6
± 15	000000000000001x	16
≥ ±16	0000000000000001xxxxxxxxxxxxxx	28

Lev-VLCT[6]		
Level	VLC	Length
± 1 to ± 32	1xxxxxx	7
± 33 to ± 64	01xxxxxx	8
...
± 449 to ± 480	0000000000000001xxxxxx	21
≥ ±481	00000000000000001xxxxxxxxxxxxxx	28

SuffixLength. Table Lev-VLCT[0] is selected for SuffixLength = 0, Table Lev-VLCT[1] for SuffixLength = 1, and so on. Lev-VLCT[0] has its own structure while the remaining VLC tables share a common structure. In all cases, when the magnitude of the level is too large, its value is stored entirely in the suffix, whose length is set to 12. As the last bit represents the sign, the maximum magnitude that CAVLC can encode is $2^{11} = 2048$ in the baseline profile [11].

Algorithm 2 [14, 32] shows how the levels are encoded. The selection of each VLC table is context adaptive, as it depends on the magnitude of the previous coded level.

Algorithm 2: CAVLC levels encoding

```

if TotalCoeff > 10 and NumT1s < 3 then
  SuffixLength ← 1
else
  SuffixLength ← 0
end if

for each level in reverse order do
  effective_level ← level
  if level is the first to encode and NumT1s < 3 then
    subtract 1 to magnitude of effective_level
  end if

  encode effective_level using Lev-VLCT[SuffixLength]

  if SuffixLength = 0 then
    SuffixLength ← 1
  end if

  if SuffixLength < 6 then
    T ← 3 × 2SuffixLength - 1
    if magnitude(level) > T then
      SuffixLength ← SuffixLength + 1
    end if
  end if
end for

```

2.1.4 TotalZeros

The symbol *TotalZeros* is the sum of all zeros preceding the last nonzero coefficient in the zigzag array. The VLC assigned to TotalZeros is obtained from a lookup table that, in the case of a 4×4 block, is selected from 15 VLC tables, whose contents are specified in Tables 9-7 and 9-8 of the H.264 standard [14]. An extract of these tables is shown in Tables 4 and 5. As it can be seen, the choice of the lookup table is done in function of the symbol TotalCoeff. If TotalCoeff is 0 or 16, TotalZeros is not encoded because it is known that all coefficients are zero or nonzero, respectively.

Table 4 Extract of the table 9-7 of the H.264 standard

TotalZeros	TotalCoeff						
	1	2	3	4	5	6	7
0	1	111	0101	0001 1	0101	0000 01	0000 01
1	011	110	111	111	0100	0000 1	0000 1
2	010	101	110	0101	0011	111	101
.....							
.....							
.....							
.....							
14	0000 0001 0	0000 00	-	-	-	-	-
15	0000 0000 1	-	-	-	-	-	-

Table 5 Extract of the table 9-8 of the H.264 standard

TotalZeros	TotalCoeff														
	8	9	10	11	12	13	14	15							
0	0000 01	0000 01	0000 1	0000	0000	000	00	0							
1	0001	0000 00	0000 0	0001	0001	001	01	1							
2	0000 1	0001	001	001	01	1	1	-							
.....															
.....															
.....															
.....															
.....															
7	001	0000 1	-	-	-	-	-	-							
8	0000 00	-	-	-	-	-	-	-							

2.1.5 Runs

The parameter *run* of a nonzero coefficient is defined as the sum of all consecutive zeros that precede it. The runs are encoded in reverse order using one of the 7 VLC tables specified in Table 9-10 of the H.264 standard [14], whose content is presented in Table 6. The selection of each VLC is done in function of the symbol run and a second parameter, called *ZerosLeft*, which is the number of zeros that remain to be encoded. *ZerosLeft* is initialized to *TotalZeros* and decreases as more runs are encoded. The runs encoding is finished in the following two cases: (1) All zeros have already been encoded. (2) The current nonzero coefficient is the last in the reverse order, which implies that the maximum value to be encoded is 14.

Table 6 Tables for runs encoding

Run	ZerosLeft						
	1	2	3	4	5	6	>6
0	1	1	11	11	11	11	111
1	0	01	10	10	10	000	110
2	–	00	01	01	011	001	101
3	–	–	00	001	010	011	100
4	–	–	–	000	001	010	011
5	–	–	–	–	000	101	010
6	–	–	–	–	–	100	001
7	–	–	–	–	–	–	0001
8	–	–	–	–	–	–	00001
9	–	–	–	–	–	–	000001
10	–	–	–	–	–	–	0000001
11	–	–	–	–	–	–	00000001
12	–	–	–	–	–	–	000000001
13	–	–	–	–	–	–	0000000001
14	–	–	–	–	–	–	00000000001

2.2 CAVLC example

In this subsection, we present an example of CAVLC encoding, corresponding to the 4×4 block of Fig. 1. As shown, the zigzag array is $\{5, 1, 0, -1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0\}$.

We assume that the two neighbouring blocks are available, and that the values of the parameters n_A and n_B are 4 and 6, respectively. Hence, $n_C = (4 + 6 + 1) \gg 1 = 5$. As $TotalCoeff = 5$ and $NumT1s = 3$, the VLC assigned to $CoeffToken$ is 1010 (see Table 1). Note that the last four nonzero coefficient have magnitude 1, but only the last three ones are taken into account.

The values of the T1s in reverse order are +1, +1, and -1. Therefore, the VLC assigned is 001.

The steps for levels encoding (see Algorithm 2 and Table 3) are the next:

1. $TotalCoeff = 5$ and $NumT1s = 3$, hence the condition $TotalCoeff > 10$ and $NumT1s < 3$ is not fulfilled and $SuffixLength$ is initialized to 0.
2. The first level in the reverse order is +1. As the condition $NumT1s < 3$ is not satisfied, the absolute value of the level is not decremented.
3. $SuffixLength$ is 0; hence, on the one hand, $Lev-VLCT[0]$ is selected and the VLC assigned to level +1 is 1. On the other hand, $SuffixLength$ is assigned the value 1.
4. As $SuffixLength$ is less than 6, on the one hand, the threshold T is calculated: $T = 3 \times 2^{SuffixLength-1} = 3 \times 2^{1-1} = 3$. On the other hand, since $magnitude(level) = 1$ and $T = 3$, the condition $magnitude(level) > T$ is not fulfilled and $SuffixLength$ is not incremented.

5. The last level to encode is +5. As SuffixLength is 1, Lev-VLCT[1] is selected and the VLC assigned to level is 000010.

TotalZeros is 2 and TotalCoeff is 5. Therefore, the value assigned to TotalZeros is 0011 (see Table 4).

The runs of coefficients 5, 1, -1, 1 and 1 are 0, 0, 1, 0 and 1, respectively. Their encoding is done as follows (see Table 6):

1. Initially, the value of ZerosLeft equals to TotalZeros, i.e., 2.
2. The first run in the reverse order is 1 and ZerosLeft is 2; therefore, the VLC assigned is 01 and the value of ZerosLeft changes to 1.
3. The second run is 0 and ZerosLeft is 1; hence, the VLC assigned is 1 and the value of ZerosLeft does not change.
4. The third run is 1 and ZerosLeft is 1, therefore the VLC assigned is 0 and the value of ZerosLeft changes to 0.
5. As all zeros have been reached, the runs encoding is finished.

Finally, taking into account the different encodings seen in this subsection, the resulting bitstream of our example is the following: 1010-001-1-000010-0011-01-1-0

3 Solution of Su et al.

The only state-of-the-art GPU-based implementation of CAVLC is the solution presented by Su et al. [38, 39], which was developed in CUDA. It satisfies the real-time processing for HDTV 720p and its throughput is 11.17 to 6.29 times higher than that of the published software encoders on DSP and multi-core platforms.

By profiling the instructions of CAVLC, Su et al. found the main factors that limit the potential of parallelism [38, 39], which are the context-based data dependence, the memory accessing dependence and the control dependence. The context-based data dependence is due to the self-adaptive feature of CAVLC. Since the value of the parameter n_C depends on n_A and n_B , it is not possible to calculate the parameter n_C of a block until the symbols TotalCoeff of the neighbouring left and top blocks have been calculated. The memory accessing dependence is caused by the inherently serial nature of variable length encoding. To determine the position of each VLC in the output bitstream it is necessary to know the lengths of the VLCs that precede it. Control dependence is caused by the existence of different processing paths in two layers: the frame layer and the block layer. In the first layer, the branches are due to the different frame types and the different components of a frame (luma DC, luma AC, chroma DC and chroma AC). In the second layer, the different processing paths are caused by the irregular characteristic of symbol data, such as whether sign trail is 1 or -1 and whether levels are zero or not.

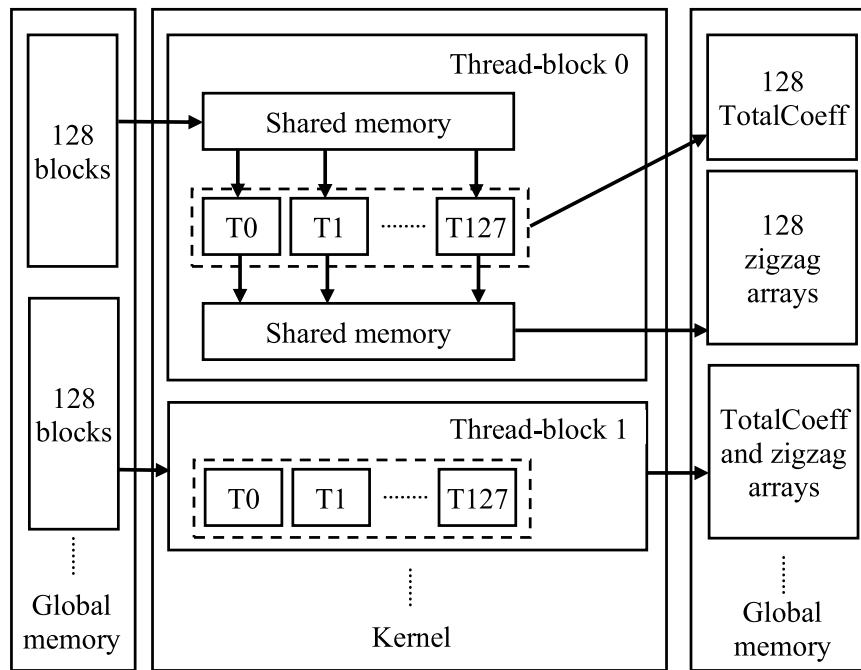


Fig. 3 Calculation of TotalCoeff and zigzag arrays in the solution of Su et al

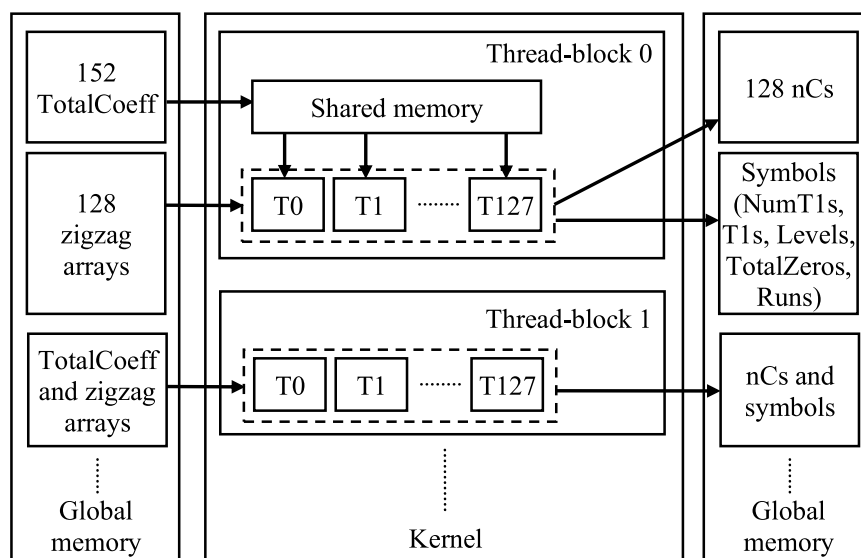


Fig. 4 Calculation of nC and other symbols in the solution of Su et al

In order to eliminate or weaken the dependencies described above, Su et al. divided the execution of CAVLC into four paths according to the four components of a frame, and the CAVLC pipeline of each path was divided into three stages: two scans, coding and lag packing.

3.1 Two scans

Two scans are employed to calculate the CAVLC symbols: a forward scan and a backward scan.

The forward scan aims at the quantized coefficients and the results include the symbols TotalCoeff and the zigzag arrays, as it is shown in Fig. 3. In this stage, each thread is assigned to deal with a block. In order to satisfy the requirement of coalesced access to global memory [22, 25], the shared memory is used as a buffer.

The backward scan is executed on the zigzag arrays generated in the first scanning and the results consist of the values of nC and the remaining CAVLC symbols (NumT1s, T1s, levels, TotalZeros and runs). In order to make better use of the local data, a frame is divided into several regions of 4×2 macroblocks. One thread-block calculates the values of nC of blocks in the same region, as it is shown in Fig. 4. The program first loads all data needed to the shared memory, then each thread visits nA and nB, where one symbol TotalCoeff can be used as either nA or nB.

3.2 Coding

For the sake of minimizing the performance loss of the target parallel CAVLC encoder due to control dependence, Su et al. proposed a component-based coding mechanism. In this method, the program codes the symbols frame by frame in order of luma DC, luma AC, chroma DC, chroma AC instead of processing the four components macroblock by macroblock. The coding method is very similar for the different types of blocks; the main difference is the use of specific lookup tables for each component. In addition, the lookup tables are firstly loaded to the shared memory to speed up the lookup operation. The configuration is similar to that of calculating the value of nC and the results are the encodings (bitstreams and bit-lengths) of each block. A memory unit of 26 short words is used to store the bitstream of a block. Figure 5 shows the organization of a thread-block for encoding the symbols.

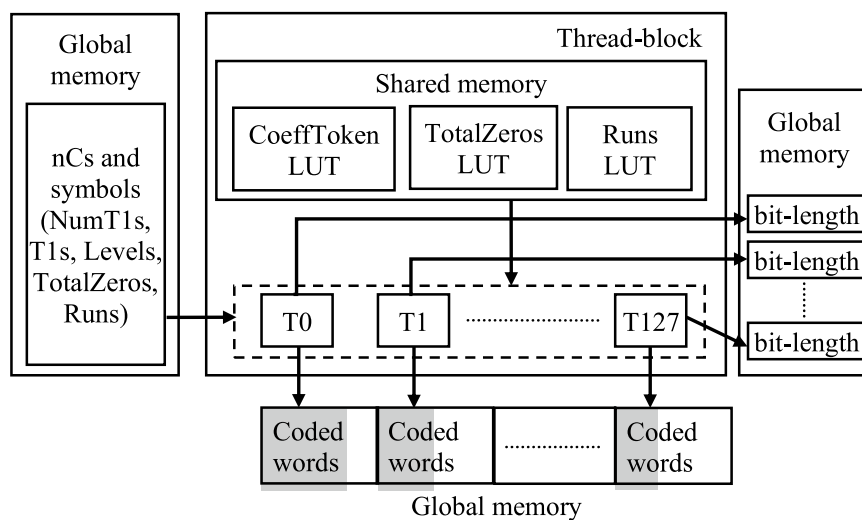


Fig. 5 Writing of encodings and bit-lengths in the solution of Su et al

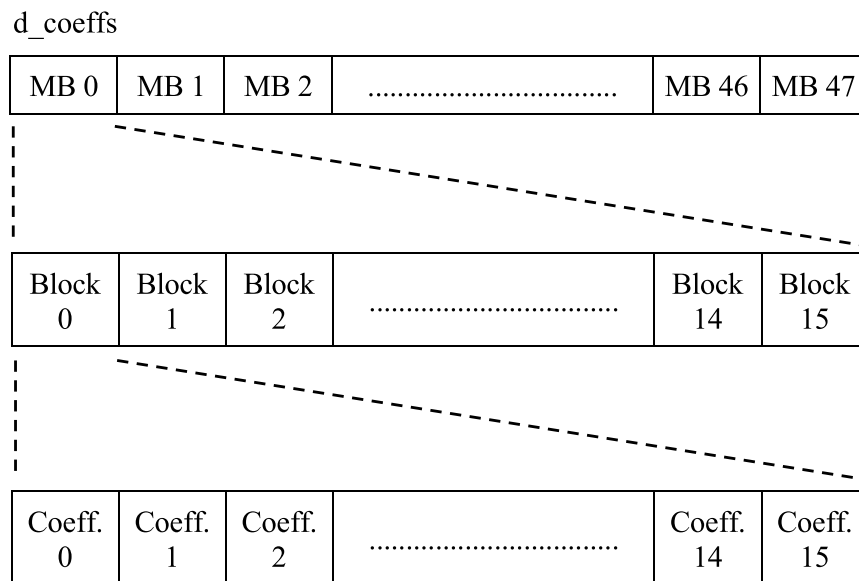


Fig. 6 Layout of CAVLCU input vector of coefficients (d_coeffs) for SQCIF format (128×96). The array is divided in as many subvectors as MBs in the frame (48 in the case of SQCIF). Each MB subvector is divided into 16 subvectors, each one corresponding to a different block of the MB

In the vector of coded words, the grey areas represent the bitstreams of the encodings, while the white regions are the unused spaces.

3.3 Lag packing

Once all the blocks are encoded, parallel writing is executed. According to the lengths of the bitstreams, the output positions are obtained and a parallel packing is performed. Thus, it can not only eliminate the constraint of accessing dependence, but it also improves the performance of writing.

4 Efficient GPU-based implementation of CAVLC (CAVLCU)

In this section, we present *CAVLCU*, our parallel implementation of CAVLC on CUDA. It is also compared with Su et al. proposal so that the achieved performance improvement can be clearly established. Our solution is built using only one CUDA kernel that has been specifically designed for encoding the luma AC blocks of a frame. The method for the remaining types of blocks (luma DC, chroma DC and chroma AC) is essentially the same, with very few variations.

The inputs of CAVLCU are the following:

- **The coefficients of the frame.** They are provided in a vector of 16-bit integers (d_coeffs), whose layout is shown in Fig. 6 for SQCIF format (128×96). As it can be seen, the array is divided in as many subvectors as macroblocks (*MBs*) in the frame (48 in the case of SQCIF format); the i -th subvector stores the coefficients of the i -th MB of the frame in the raster scan order (i.e., from left to right and

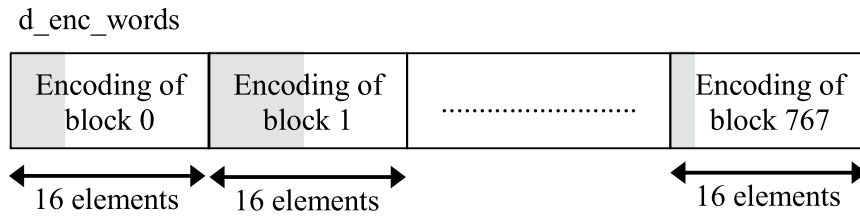


Fig. 7 Layout of CAVLCU output vector of CAVLC encodings (*d_enc_words*) for SQCIF format (128×96). The array is divided in as many subvectors of size 16 as blocks in the frame (768 in the case of SQCIF). The *i*-th subvector is used for storing the encoding of the *i*-th block of the frame. The grey areas correspond to the CAVLC encodings, whose lengths are variable

Region 0				Region 1				Region 2			
MB 0	MB 1	MB 2	MB 3	MB 4	MB 5	MB 6	MB 7	MB 8	MB 9	MB 10	
	Region 3				Region 4				Region 5		
MB 11	MB 12	MB 13	MB 14	MB 15	MB 16	MB 17	MB 18	MB 19	MB 20	MB 21	
		Region 6				Region 7					
MB 22	MB 23	MB 24	MB 25	MB 26	MB 27	MB 28	MB 29	MB 30	MB 31	MB 32	
Region 8			Region 9				Region 10				
MB 33	MB 34	MB 35	MB 36	MB 37	MB 38	MB 39	MB 40	MB 41	MB 42	MB 43	
Region 11				Region 12				Region 13			
MB 44	MB 45	MB 46	MB 47	MB 48	MB 49	MB 50	MB 51	MB 52	MB 53	MB 54	
	Region 14				Region 15				Region 16		
MB 55	MB 56	MB 57	MB 58	MB 59	MB 60	MB 61	MB 62	MB 63	MB 64	MB 65	
		Region 17				Region 18					
MB 66	MB 67	MB 68	MB 69	MB 70	MB 71	MB 72	MB 73	MB 74	MB 75	MB 76	
Region 19			Region 20				Region 21				
MB 77	MB 78	MB 79	MB 80	MB 81	MB 82	MB 83	MB 84	MB 85	MB 86	MB 87	
Region 22				Region 23				Region 24			
MB 88	MB 89	MB 90	MB 91	MB 92	MB 93	MB 94	MB 95	MB 96	MB 97	MB 98	

Fig. 8 Partition of a QCIF frame (176×144) in regions of size 4

from top to bottom). At the same time, each MB subvector is divided into 16 subvectors, each one corresponding to a different block of the MB; the *i*-th subvector stores the 16 coefficients of the *i*-th 4×4 block of the MB. Both blocks and coefficients are provided in the raster scan order as well.

- **The prediction modes of the MBs.** They are supplied in a vector of 8-bit integers (*d_MB_pred_modes*), where the *i*-th element is assigned to the *i*-th MB of the frame.
- **The slice IDs of the MBs.** They are provided in a vector of 16-bit integers (*d_MB_slices*), where the *i*-th element is assigned to the *i*-th MB of the frame.

Similarly, the outputs of CAVLCU are the following:

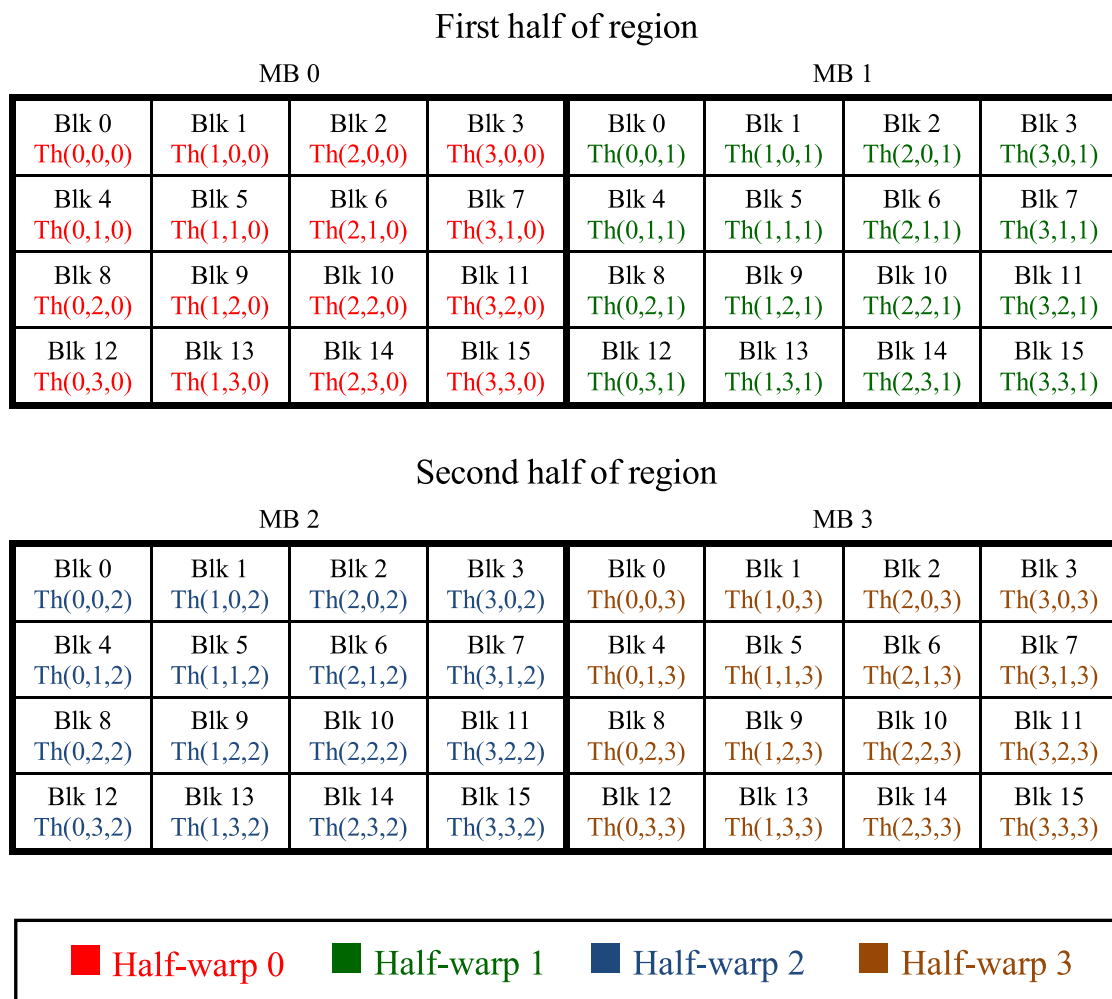


Fig. 9 Mapping of region data to elements of a thread-block in the case of region 7 of Fig. 8. The first MB of the region is assigned to the first half-warp of the thread-block, the second MB to the second half-warp, and so on. For each MB, the first block is assigned to the first thread of the corresponding half-warp, the second block to the second thread, and so on

- **The encodings of the blocks.** They are written in a vector of 32-bit integers, d_enc_words , where the i -th subvector of size 16 (BLK_ENC_SIZE) is used for storing the encoding of the i -th block of the frame, as Fig. 7 illustrates for SQCIF format.
- **The binary lengths of the encodings.** They are stored in a vector of 16-bit integers, d_enc_lens , where the i -th element is assigned to the i -th block of the frame.

As illustrated for a QCIF frame (176×144) in Fig. 8, CAVLCU divides a frame into equally-sized groups of consecutive MBs in the raster scan order, which will be referred to as *regions*. The execution configuration of the kernel uses a one-dimensional grid with as many thread-blocks as regions in the frame (NUM_REG); the i -th thread-block of the grid processes the i -th region of the frame. The dimensions of the thread-blocks are $4 \times 4 \times REG_SIZE$, where REG_SIZE is the number of MBs of each region. As it is shown in Fig. 9 for the region 7 of Fig. 8, the i -th MB of the

region is assigned to the i -th half-warp of the thread-block, and the i -th block of a MB is encoded by the i -th thread of the corresponding half-warp.

Algorithm 3 shows the pseudocode of CAVLCU kernel. The parameters NUM_MB , NUM_BLK and NUM_COEFF represent, respectively, the number of MBs, 4×4 blocks and coefficients of the frame; on the other hand, NUM_WORD_ENC is the number of 32-bit words used for storing the CAVLC encodings of the frame, whose value is the product of NUM_BLK by BLK_ENC_SIZE . Each thread performs the next steps. First, it calculates the indexes of the block to be encoded and the MB to which the block belongs. Second, it reads the coefficients of the block, and the prediction mode and slice ID of the MB. Third, it sorts the block in zigzag order to get the zigzag array. Fourth, it calculates a set of symbols from the zigzag array. Fifth, it calculates the parameter nC . Sixth, it uses the symbols and the parameter nC to encode the block.

Algorithm 3: CAVLCU algorithm

```

kernel CAVLCU(// outputs
    d_enc_words[NUM_WORD_ENC],
    d_enc_lens[NUM_BLK],
    // inputs
    d_coeffs[NUM_COEFF],
    d_MB_pred_modes[NUM_MB],
    d_MB_slices[NUM_MB]
)
    // Calculate the block and MB indexes
    blk_idx ← calculate_thread_ID()
    MB_idx ← blk_idx / 16

    // Read the coefficients of the block
    longlong4 block_vec ← ((longlong4 *)d_coeffs)[blk_idx]

    // Read the prediction mode and the slice ID of the MB
    MB_pred_mode ← d_MB_pred_modes[MB_idx]
    MB_slice ← d_MB_slices[MB_idx]

    // Sort the block in zigzag order
    short zz_array[16]
    sort_in_zigzag_order(zz_array, block_vec)

    // Calculate a set of symbols from the zigzag array
    calculate_symbols(// outputs
        TotalCoeff, NumT1s, T1s,
        ZigzagLevelsMask, ZigzagArrayMask,
        // inputs
        zz_array, MB_pred_mode
    )

    nC ← manage_nC(TotalCoeff, MB_slice, blk_idx, MB_idx)

    encode_block(// outputs
        d_enc_words, d_enc_lens,
        // inputs
        blk_idx,
        TotalCoeff, NumT1s, T1s,
        ZigzagLevelsMask, ZigzagArrayMask,
        nC
    )
end kernel

```

4.1 Calculation of block and MB indexes

As the i -th block of the frame is processed by the i -th thread of the grid, the index of the block equals to the thread ID in the grid, whose value is the following:

$$\begin{aligned} & blockIdx.x \times (blockDim.x \times blockDim.y \times blockDim.z) + \\ & + threadIdx.z \times blockDim.x \times blockDim.y + \\ & + threadIdx.y \times blockDim.x + threadIdx.x \end{aligned}$$

Since $blockDim.x = blockDim.y = 4$ and $blockDim.z = REG_SIZE$, the index of the block is calculated using the following expression:

$$\begin{aligned} & 16 \times REG_SIZE \times blockIdx.x + 16 \times threadIdx.z + \\ & + 4 \times threadIdx.y + threadIdx.x \end{aligned}$$

As a MB is composed of 16 blocks, the index of the MB is obtained by dividing the block index by 16.

4.2 Coefficients reading

Each thread reads the 16 coefficients of its block through one vectorized access using the built-in vector type *longlong4* [25], whose definition is shown in Algorithm 4. Since the sizes of types *long long int* and *short* are 8 and 2 bytes, respectively, each member of the variable *block_vec* contains 4 coefficients of the current block; as shown in Table 7, the member *x* contains the first 4 coefficients, the member *y* the next 4, and so on. Vectorized loads are an important CUDA optimization because they increase bandwidth and reduce both instruction count and latency [18].

In contrast, the solution of Su et al. uses the shared memory as a buffer to fulfill the requirement of coalesced global memory accesses recommended in CUDA literature [22, 25]. Since the maximum amount of shared memory per multiprocessor is 48 KB for GPUs with compute capability less than 3.7 [24] and the size of a block is 32 bytes, the occupancy is penalized in these architectures. For example, if the number of threads per thread-block is 128, the theoretical occupancy is reduced to 75% (3.x) or 67% (2.x) [24].

Table 7 Mapping of block coefficients to *longlong4* members

Member	Bits			
	0–15	16–31	32–47	48–63
<i>x</i>	Coeff. 0	Coeff. 1	Coeff. 2	Coeff. 3
<i>y</i>	Coeff. 4	Coeff. 5	Coeff. 6	Coeff. 7
<i>z</i>	Coeff. 8	Coeff. 9	Coeff. 10	Coeff. 11
<i>w</i>	Coeff. 12	Coeff. 13	Coeff. 14	Coeff. 15

Algorithm 4: CUDA built-in vector types
longlong4 and uchar2

```

struct __device_builtin__ __builtin_align__(16) longlong4
{
    long long int x, y, z, w;
};

struct __device_builtin__ __align__(2) uchar2
{
    unsigned char x, y;
};

```

4.3 Zigzag sorting

The coefficients of a block are extracted from the variable `block_vec` and are written in the private array `zz_array` in zigzag order. This operation is based on the mapping shown in Table 7. The performance of this operation is high for the following reasons. First, there are no dependencies between the different coefficient extractions; hence, the degree of instruction-level parallelism is high. Second, each coefficient extraction is performed with few operations of high throughput (two binary shifts and a cast). Third, `zz_array` is placed in register space [22] because (1) it is small; (2) it is indexed with constant quantities, and (3) the kernel does not use more registers than available.

The solution of Su et al., after loading the blocks in shared memory, write them back to global memory in zigzag order; the coefficients will be read later again for calculating the CAVLC symbols. Conversely, CAVLCU executes the zigzag sorting in a much more efficient way, as it only consists of few high throughput operations with high degree of ILP reading and writing in the register space and saving costly memory global accesses.

4.4 Calculation of the symbols

Algorithm 5 shows the pseudocode for calculating the following symbols, which will be used for encoding the current block:

- The CAVLC symbols `TotalCoeff`, `NumT1s` and `T1s`.
- A 16-bit binary mask (*ZigzagArrayMask*) which represents the structure of the zigzag array and hence implicitly the CAVLC symbols `TotalZeros` and `runs`. If the *i*-th coefficient of the zigzag array is non-zero, the *i*-th most significant bit of the mask is 1; otherwise, this bit is 0. In the example of Fig. 1, the value of the mask is 1101101000000000.
- A second 16-bit binary mask (*ZigzagLevelsMask*) which represents the structure of the zigzag array excluding the trailing ones. In the example of Fig. 1, the value of the mask is 1100000000000000.

Algorithm 5: Calculation of symbols

```

#define MASK_OF_COEFF(coeff_idx) (1 << (15 - coeff_idx))

function calculate_symbols(// outputs
    TotalCoeff, NumT1s, T1s,
    ZigzagLevelsMask, ZigzagArrayMask,
    // inputs
    zz_array[16], MB_pred_mode
)
    // Initialize symbols
    NumT1s ← 0
    T1s ← 0
    ZigzagLevelsMask ← 0
    ZigzagArrayMask ← 0

    // Process AC coefficients
    for coeff_idx = 15 to 1 step -1 do
        if zz_array[coeff_idx] <> 0 then
            update_symbols(// outputs
                NumT1s, T1s,
                ZigzagLevelsMask, ZigzagArrayMask,
                // inputs
                zz_array[coeff_idx],
                MASK_OF_COEFF(coeff_idx)
            )
        end if
    end for

    if MB_pred_mode = INTRA_16×16 then
        ZigzagArrayMask ← (ZigzagArrayMask << 1)
    else
        // Process DC coefficient
        if zz_array[0] <> 0 then
            update_symbols(// outputs
                NumT1s, T1s,
                ZigzagLevelsMask, ZigzagArrayMask,
                // inputs
                zz_array[0],
                MASK_OF_COEFF(0)
            )
        end if
    end if

    TotalCoeff ← __popc(ZigzagBlock_Mask)
end function

```

Algorithm 6: Symbols updating from a non-zero coefficient

```

function update_symbols(// outputs
                      NumT1s, T1s,
                      ZigzagLevelsMask, ZigzagArrayMask,
                      // inputs
                      coeff, mask_of_coeff
                      )
if abs(coeff) = 1 and NumT1s < 3 and ZigzagLevelsMask = 0 then
  // The coefficient is a trailing one
  T1s ← (T1s << 1) | (coeff = 1 ? 0 : 1)
  NumT1s ← NumT1s + 1
else
  // The coefficient is a level
  ZigzagLevelsMask ← ZigzagLevelsMask | mask_of_coeff
end if

ZigzagArrayMask ← ZigzagArrayMask | mask_of_coeff
end function

```

Each thread performs the next steps. First, it initializes all the symbols to 0. Second, for each nonzero AC coefficient stored in *zz_array* (i.e., all but the first), from the last to the first, it updates all the symbols except *TotalCoeff* performing the steps presented in Algorithm 6. Third, if the prediction mode of the current MB is not Intra 16×16, it processes the DC coefficient (i.e., the first) in the same way as in step 2. Otherwise, it ignores the DC coefficient and left-shift the symbol *ZigzagArrayMask* one bit, as only the subblock formed by the AC coefficients must be considered. Fourth, it calculates *TotalCoeff* from *ZigzagArrayMask* using the CUDA function `__popc` [23], which counts the number of bits that are set to 1 in a 32 bit integer. The throughput of `__popc` is high as it compiles to a single instruction [25].

In the solution of Su et al., each thread iterates two times over the coefficients of a block for calculating its CAVLC symbols: *TotalCoeff* in the first iteration and the remaining ones (*NumT1s*, *T1s*, *levels*, *TotalZeros*, *runs*) in the second. All the symbols are written in global memory and later read for transferring them between the corresponding kernels. CAVLCU optimizes significantly this process for the following reasons. First, it iterates only one time over the coefficients of a block for calculating the necessary symbols. Second, the number of symbols processed in the loop is reduced to only 4 integers: *NumT1s*, *T1s*, *ZigzagLevelsMask* and *ZigzagArrayMask*. Third, as shown in Algorithm 6, the update of the symbols in each loop iteration is performed very efficiently, as it only requires two OR operations for the symbols *ZigzagLevelsMask* and *ZigzagArrayMask*, an addition for *NumT1s* and a binary left shift, an OR operation and a comparison for *T1s*. Fourth, our algorithm saves read/write global memory operations performed by the solution of Su et al. as transferring symbols between kernels is not required.

Fig. 10 Reading of parameter nA using the CUDA function `__shfl_up`

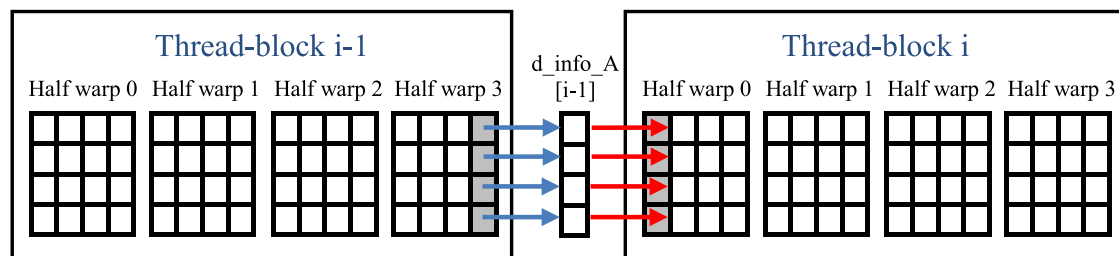
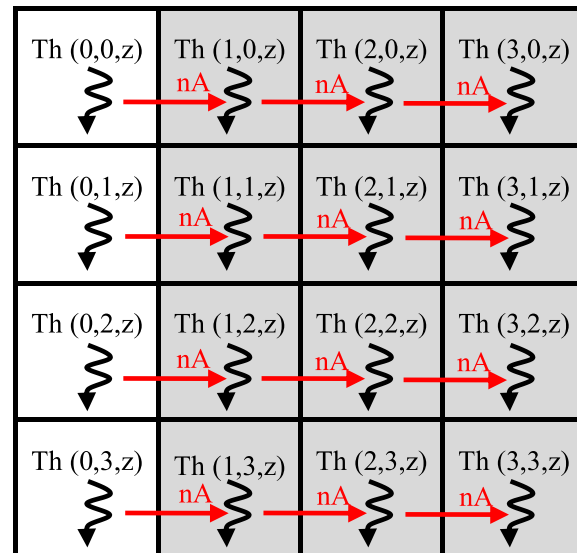


Fig. 11 Transmission of parameter nA through global memory

4.5 Calculation of parameter nC

According to the method described in Subsection 2.1.1, each thread calculates the parameter nC of its block from the information associated to the left and top neighbouring blocks ($info_A$ and $info_B$, respectively). Each block information is composed of the symbol TotalCoeff and the slice ID. The symbols TotalCoeff of the left and top blocks are the parameters nA and nB , respectively. The slice ID of the left and top blocks will be denoted as $SliceID_A$ and $SliceID_B$, respectively.

Each thread (x, y, z) gets $info_A$ as follows. If the current block is not in the first column of its MB, nA is read from the left thread $(x - 1, y, z)$ using the CUDA function `__shfl_up` [25], as shown in Fig. 10. As both left and current block are in the same MB, $SliceID_A$ is the slice ID of the current MB. If the current block is in the first column of the first MB of a region, $info_A$ is read from an intermediate array in global memory (d_info_A) of dimensions $NUM_REG \times 4$. As illustrated in Fig. 11, each thread $(0, y, 0)$ of a thread-block i reads $info_A$ from the element $d_info_A[i - 1][y]$, which is written by the thread $(3, y, REG_SIZE - 1)$ of the thread-block $i - 1$. If the current block is in the first column of the second or posterior MB of a region, $info_A$ is read from an intermediate array in shared memory (s_info_A) of dimensions $REG_SIZE \times 4$. As illustrated in Fig. 12, each thread $(0, y, z)$ with $z > 0$ of a thread-block reads $info_A$ from the element $s_info_A[z - 1][y]$, which is written by the thread $(3, y, z - 1)$.

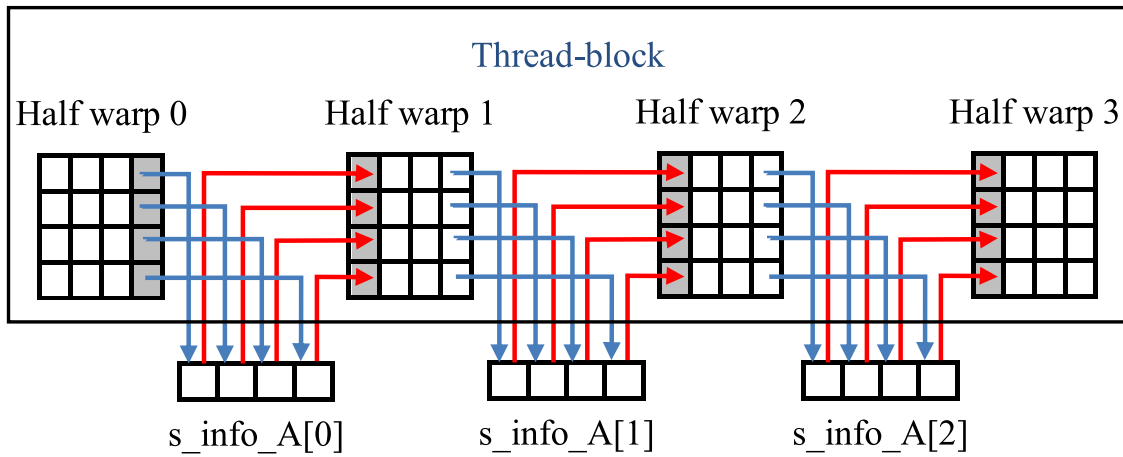
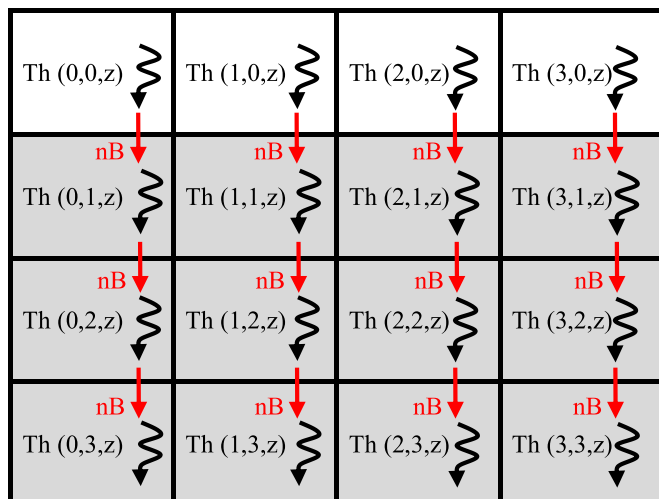


Fig. 12 Transmission of parameter nA through shared memory

In a similar way, each thread (x, y, z) gets `info_B` as follows. If the current block is not in the first row of its MB, `nB` is read from the top thread $(x, y - 1, z)$ using the CUDA function `__shfl_up` [25], as shown in Fig. 13. As both top and current block are in the same MB, `SliceID_B` is the slice ID of the current MB. If the current block is in the first row of its MB, `info_B` is read from an intermediate array in global memory (`d_info_B`) of dimensions $NUM_MB_VER \times NUM_MB_HOR \times 4$, where `NUM_MB_VER` and `NUM_MB_HOR` are the number of MBs in the vertical and horizontal dimensions of the frame, respectively. As illustrated in Fig. 14, each thread $(x, 0, z)$ of a thread-block reads `info_B` from the element `d_info_B[r - 1][c][x]`, where r and c are, respectively, the row and the column of the current MB. Each element `d_info_B[r][c][x]` is written by the thread $(x, 3, z)$ of the half-warp that processes the MB in row r and column c .

Algorithm 7 shows the pseudocode for managing the parameter `nC`. Each thread performs the next steps. First, it represents the necessary information of the current block (the symbol `TotalCoeff` and the slice ID of its MB) in a compact way using a 32-bit integer (`info`), where the 5 least significant bits store `TotalCoeff`, the sixth

Fig. 13 Reading of parameter nB using the CUDA function __shfl_up



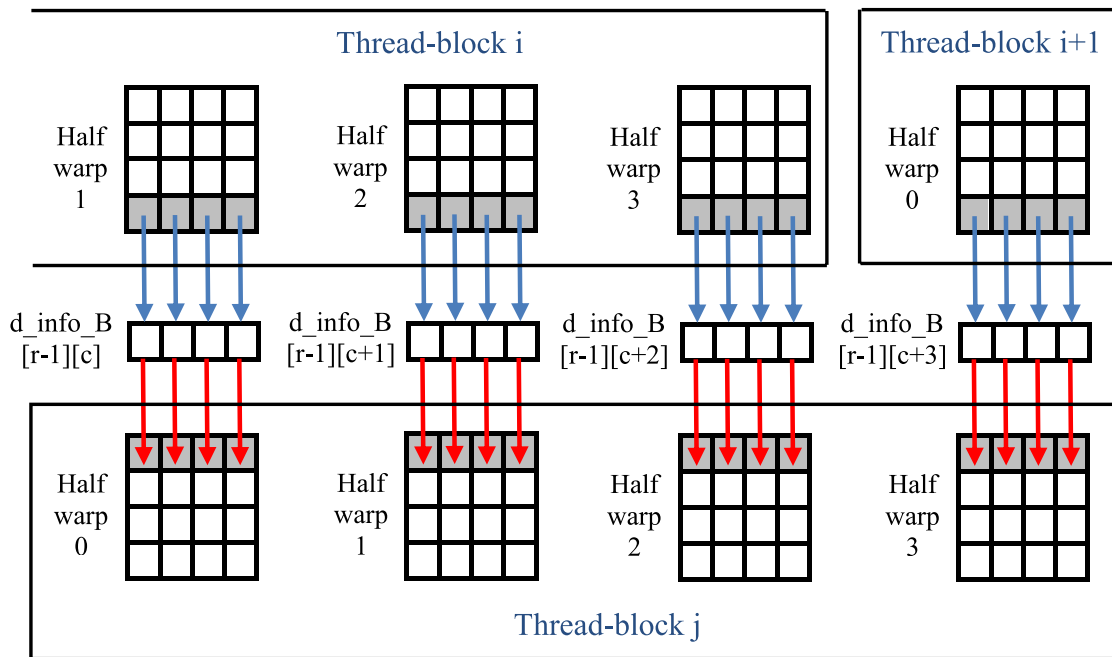


Fig. 14 Transmission of parameter nB through global memory

least significant bit is set to 1, which ensures $info$ is nonzero, and the 16 most significant bits store the slice ID. Second, it calculates the indexes of the row and the column of the MB in the frame. Third, it proceeds, writes $info$ in the intermediate arrays as described above. Fourth, it synchronizes with other threads of the block for ensuring the array s_info_A contains the correct values. Fifth, it gets nA and nB as explained above. If a neighbouring block is unavailable, the corresponding reading function ($read_nA$ or $read_nB$) returns -1 . Sixth, it calculates nC from nA and nB using the method shown in Table 2.

Algorithm 7: Management of parameter nC

```

function manage_nC(TotalCoeff, MB_slice, blk_idx, MB_idx)
  info ← (MB_slice << 16) + 64 + TotalCoeff
  MB_row ← MB_idx / NUM_MB_HOR
  MB_col ← MB_idx % NUM_MB_HOR

  write_info_B_of_right_MB(info, MB_row, MB_col)
  write_info_A_of_right_MB(info, MB_col)

  __syncthreads()

  nB ← read_nB(TotalCoeff, MB_slice, MB_row, MB_col)
  nA ← read_nA(MB_col, TotalCoeff, MB_slice)

  if nA = -1 and nB = -1 then
    nC ← 0
  else if nB = -1 then
    nC ← nA
  else if nA = -1 then
    nC ← nB
  else
    nC ← (nA + nB + 1) >> 1
  end if

  return nC
end function

```

As in our previous works [7, 8], the thread-block synchronization mechanism proposed by Yan et al. [47] is used for synchronizing the reads with the writes in global memory. In this case, it is applied on both horizontal (d_info_A) and vertical (d_info_B) dimensions and the reads are performed using atomic operations. The elements of d_info_A and d_info_B are initialized to 0 statically. Since all the values written are nonzero (due to the fact that the sixth least significant bit is set to 1), the read of each element is performed executing the CUDA atomic function *atomicExch* [25] repeatedly until a nonzero value is returned. Additionally, the use of this function restores the stored value to 0, which allows subsequent uses of the intermediate arrays in global memory, and avoids getting old cached values.

As static initialization of variables in shared memory is illegal in CUDA, a different synchronization mechanism is used in the accesses to s_info_A . In this case, the CUDA intrinsic function *__syncthreads()* guarantees that each element is not read until its value has been written. On the other hand, the use of the keyword *volatile* in the declaration of the array s_info_A ensures any reference to this variable compiles to an actual memory read or write instruction [25].

CAVLCU reduces significantly the number of global memory accesses with respect to the solution of Su et al. for the following reasons. First, in our solution, each thread-block, on the one hand, only writes in global memory the symbols TotalCoeff of the last column and the last row of its region and, consequently, on the other hand, only reads from global memory the parameters nA and nB of the first column and the first row, respectively. In contrast, the solution of Su et al. writes all the symbols TotalCoeff of the frame in global memory and each thread-block not only reads from global memory the parameters nA and nB of the first column and

the first row of its region but also all the symbols `TotalCoeff` of the region. Second, in the approach of Su et al., once the parameters `nC` are calculated, they are written in global memory to be read in the coding stage. Therefore, CAVLCU saves two operations in global memory for writing and reading all the parameters `nC` of a frame.

4.6 Block encoding

The first action of this stage is to call the CUDA warp synchronization function `__syncwarp()` [25] to force reconvergence. This prevents the independent thread scheduling of modern architectures (Volta and later) from increasing the number of global memory writes.

Each thread i of the grid writes the encoding of its block in the subvector i of `BLK_ENC_SIZE` elements (`d_blk_enc`) of `d_enc_words` (see Fig. 7), and the bit-length of the encoding in the element i of `d_enc_lens`.

The block encoding is constructed in the way specified in Sect. 2. As the VLCs assigned to the CAVLC symbols are obtained, their bits are concatenated in a 32-bit variable (`word_val`) and their lengths added in a second 32-bit variable (`word_len`) while the bit-length of the resulting encoding is less than or equal to 32. When the last condition is not satisfied, the first 32 bits of the resulting encoding are written in the corresponding element of `d_blk_enc`, and the value and length of the remaining encoding are stored in `word_val` and `word_len`, respectively. The process continues until all the VLCs are written. The bit-length of the encoding is written in the element i of `d_enc_lens`. Its value is calculated using the following expression, where `word_idx` is the index of the last accessed position of `d_blk_enc`:

$$\text{word_idx} \times 32 + \text{word_len}$$

The lookup tables of the symbols `CoeffToken`, `TotalZeros` and `runs` are stored in arrays in global memory, which are initialized statically. The base type is the CUDA intrinsic vector type `uchar2` [25], whose definition is presented in Algorithm 4. The members x and y represent, respectively, the bit-length and the value of a variable-length code. The CUDA function `__ldg` [25] is used for caching the reads in the read-only data cache [22]. In contrast, the solution of Su et al. uses the shared memory for caching the lookup tables. Both memory systems have a small latency but the use of the read-only data cache saves a synchronization barrier and additional instructions for caching the lookup tables programmatically.

The VLCs of the levels are calculated using the method for encoding levels without lookup tables presented by Hoffman et al. [11]. The values of the levels are extracted from the variable `block_vec` using the positions stored in the symbol `ZigzagLevelsMask`.

The symbols `TotalZeros` and `runs` are obtained from the positions in reverse order of the nonzero coefficients in the zigzag array. As illustrated in Table 8 for the example of Fig. 1, the coefficients positions are 0 for the last coefficient, 1 for the penultimate coefficient, and so on. In the case of `TotalZeros` the following expression is used:

Table 8 Coefficients positions of zigzag array of Fig. 1

Zigzag array	5	1	0	-1	1	0	1	0	0	0	0	0	0	0	0	
Coefficients positions	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

16 - TotalCoeff - last_coeff_pos where *last_coeff_pos* is the position of the last nonzero coefficient. In our example, TotalCoeff is 5 and last_coeff_pos is 9; therefore, TotalZeros is 16 - 5 - 9 = 2. The symbol run associated to each nonzero coefficient is calculated using the next expression:

$$\text{prev_coeff_pos} - \text{coeff_pos} - 1$$

where *coeff_pos* and *prev_coeff_pos* are the positions of the current and immediately previous nonzero coefficients, respectively. Table 9 shows the calculation of the runs for our example.

The positions of the nonzero coefficients are obtained from ZigzagArrayMask and ZigzagLevelsMask in reverse order (i.e., from the last nonzero coefficient to the first) calling the CUDA function *__ffs* [23], which finds the position of the least significant bit set to 1 in a 32 bit integer. After each function call, the last bit of the corresponding mask is set to 0 using the next expression, where *~* and *<<* are the bitwise operators AND, NOT and left shift, respectively,

$$\text{ZigzagArrayMask} \& \sim(1 \ll (\text{coeff_pos} - 1))$$

5 Experimental evaluation

We have evaluated CAVLCU and compared it to the only existing state-of-the-art GPGPU implementation of CAVLC, which is the solution proposed by Su et al. [38, 39]. It will be referred to as *CAVLC_SU* in this section. We implemented CAVLC_SU from scratch following the description of the algorithm given by their authors [38, 39] and their support through private communication with Huayou Su.

We used two GPUs to test the algorithms, a GeForce GTX 970 (Maxwell architecture with compute capability 5.2) and a GeForce RTX 2080 (Turing architecture with compute capability 7.5).

In order to compare CAVLCU with CAVLC_SU, we measured the execution time, the number of global transactions and the number of executed instructions for the first 50 frames of the video sequences City (QCIF), Mother and Daughter (CIF), and Ducks take off (720p) [44]. Each test was performed with a GOP length of 10 and for 11 values of the quantization parameter (QP) between 0 and 50. The number of threads per thread-block was 128 in all cases; hence, the value of the parameter REG_SIZE of CAVLCU was 8.

Figures 15, 16 and 17 present the execution times in milliseconds and Table 10 the minimum, maximum and average values of CAVLCU speedup with respect to CAVLC_SU. As it can be seen, the results on both Maxwell and Turing architectures showed that our algorithm clearly outperforms the solution of Su et al. [38,

Table 9 Calculation of symbols Runs of zigzag array of Fig. 1. The parameters `coeff_pos` and `prev_coeff_pos` represent the positions of the current and immediately previous nonzero coefficients, respectively

Coefficient	<code>coeff_pos</code>	<code>prev_coeff_pos</code>	Run
1	9	11	$11-9-1 = 1$
1	11	12	$12-11-1 = 0$
-1	12	14	$14-12-1 = 1$

Table 10 Execution time speedup of CAVLCU with respect to CAVLC_SU

Video clip	Maxwell GPU			Turing GPU		
	Minimum	Maximum	Average	Minimum	Maximum	Average
City	2.7	3.6	3.3	4.2	6.2	5.2
Mother and Daughter	2.5	3.5	3.3	3.9	5.4	4.8
Ducks take off	3.3	5.4	4.1	3.0	6.7	5.1

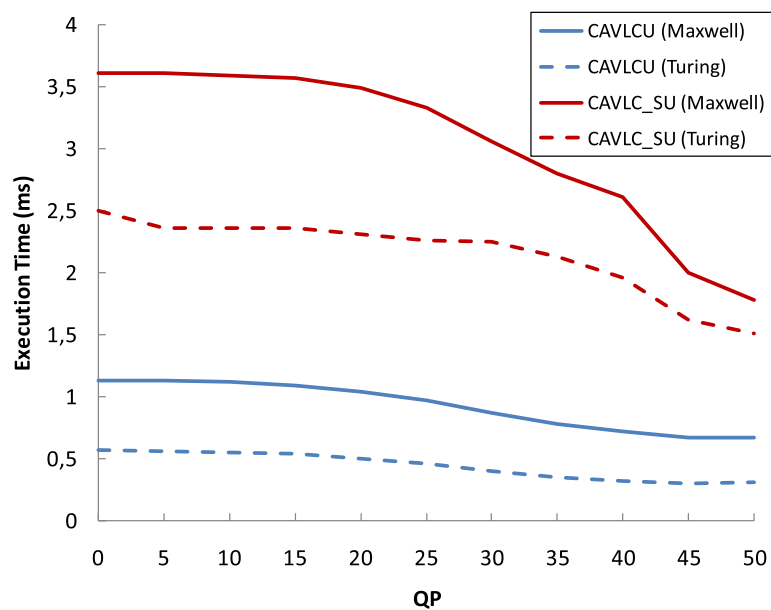


Fig. 15 Execution Time (ms) versus Quality Parameter (QP) for video clip "City" (QCIF)

39], since CAVLCU is between 2.5 and 5.4 faster than CAVLC_SU on the first architecture and between 3.0 and 6.7 on the second.

As the main improvement of our implementation is given by the reduction of global memory access, Table 11 compares the number of these memory operations for CAVLCU and CAVLC_SU. It shows that our implementation reduces in a 75.70% the number of global memory transactions in Maxwell architecture and a 65.86% in Turing architecture. Thus, since CAVLCU is built using only one kernel, it saves many of the CAVLC_SU global memory accesses required

Fig. 16 Execution Time (ms) versus Quality Parameter (QP) for video clip "Mother and Daughter" (CIF)

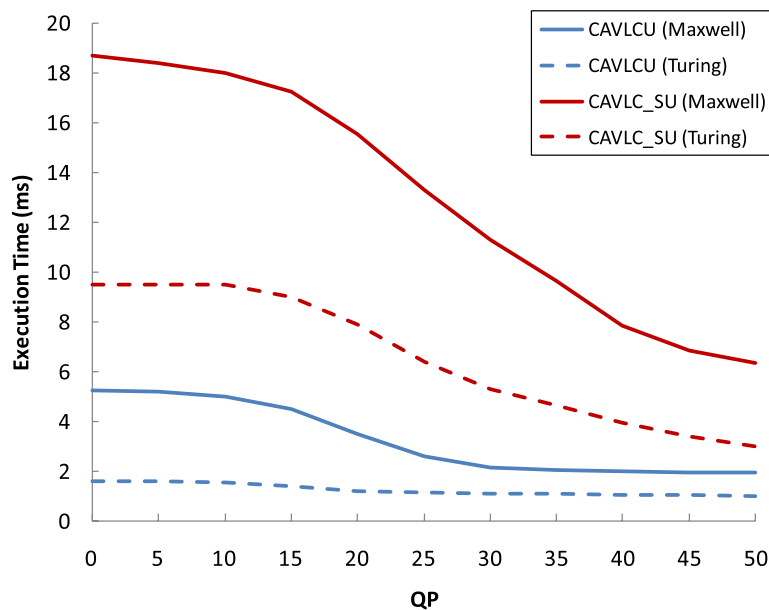
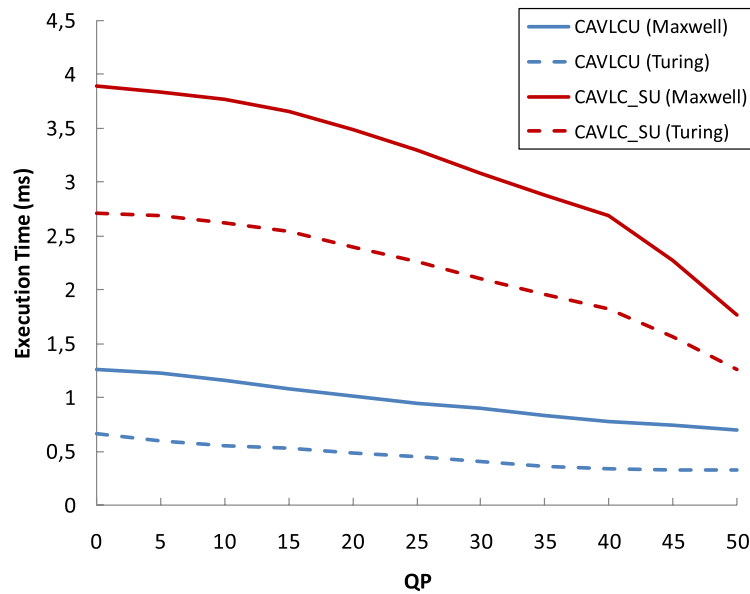


Fig. 17 Execution Time (ms) versus Quality Parameter (QP) for video clip "Ducks take off" (720p)

for communicating intermediate results among kernels for the following reasons. First, the forward scan of CAVLC_SU reads all the coefficients of the frame from global memory and write them back ordered in zigzag; later, the coefficients are read again by the backward scan. CAVLCU reads the coefficients only once (in an efficient way through vectorized accesses) and does not need to write them back. Second, the forward scan of CAVLC_SU writes the symbols TotalCoeff in global memory for its posterior reading by the backward scan; in a similar way, the backward scan writes the remaining symbols in global memory for its posterior reading in the coding stage. In contrast, CAVLCU holds all the symbols in the register space and does not need to process them in global memory. Third,

Table 11 Number of global transactions and executed instructions

Video clip	Solution	Maxwell GPU			Turing GPU		
		Global Load Transac.	Global Store Transac.	Executed Instructions	Global Load Transac.	Global Store Transac.	Executed Instructions
City	CAVLC_SU	22,479,009	5,108,650	84,451,824	7,001,064	5,108,650	84,782,589
	CAVLCU	5,208,773	1,489,900	36,641,924	2,731,105	1,489,900	38,980,702
	Improvement	4.32	3.43	2.30	2.56	3.43	2.17
Mother and Daughter	CAVLC_SU	71,176,225	15,051,190	235,433,090	21,833,529	15,051,190	235,679,200
	CAVLCU	16,723,445	4,683,296	96,217,749	8,558,844	4,683,296	102,257,746
	Improvement	4.26	3.21	2.45	2.55	3.21	2.30
Ducks take off	CAVLC_SU	719,933,101	181,420,862	2,754,450,830	224,018,729	181,420,862	2,767,884,867
	CAVLCU	163,912,274	54,654,524	1,226,364,124	83,021,412	54,654,524	1,301,713,005
	Improvement	4.39	3.32	2.25	2.70	3.32	2.13

the backward scan of CAVLC_SU writes all the parameters nC in global memory for its posterior reading in the coding stage. In contrast, CAVLCU uses the thread-block synchronization mechanism of Yan et al. [47] for transmitting only the parameters nC that are strictly necessary in an efficient way.

As seen in Table 11, CAVLCU improves the number of executed instructions by a factor higher than 2, due to the greater simplicity of our algorithm, which uses vectorized loads for reading the blocks, saves intermediate results transmission among kernels and, unlike CAVLC_SU, does not cache the lookup tables in shared memory.

6 CAVLC applications

Over the years, many adaptations of CAVLC have been proposed in different fields, like data encryption [19, 40–42] and information hiding [16, 17, 45, 46].

Mian et al. [19] proposed a technique which consists in encrypting codeword indexes and looking up codeword tables to determine the new codewords according to the encrypted indexes. They embedded encryption into the process of encoding TotalCoeffs, 4×4 block TotalZeros, chroma DC 2×2 block TotalZeros and runs. Experiments showed that the algorithm is able to provide compromise between security and complexity, and has little effect on compression performance.

Wang et al. [42] demonstrated that two fast selective encryption methods for CAVLC and CABAC [34–36] are not as efficient as only encrypting the sign bits of nonzero coefficients. As a much stronger scrambling effect can be achieved encrypting the sign bits of intra prediction modes and motion vectors, they proposed a tunable encryption method based on these three ways of encryption. Experiments showed that this method has null or very little impact on compression performance. It can run in real-time and its computational cost is minimal. It is secure against the replacement attack when all three control factors are set to one.

Tabash and Izharuddin [40] presented a technique based on Baker's map, a two-dimensional chaotic map, which is used to design a pseudorandom number generator (PRNG). The proposed PRNG is used to encrypt the sign of transformed coefficients, the codewords of runs and the pattern of trailing ones. Experiments showed good encryption results, where the visual information was successfully encrypted. The proposed method is secure against common attacks and has low computational requirements.

Kim et al. [16] proposed a fragile watermarking scheme where the hidden information is embedded in the first sign bit of the CAVLC trailing ones encodings. The bitrate of the watermarked video remains the same and the PSNR is higher than 43 dB.

Liao et al. [17] presented an information hiding algorithm which follows the next steps: (1) Generate random sequences based on chaotic maps to select the block positions for embedding the data. (2) Assign the i -th hidden bit to the parity of the number of trailing ones of the i -th block, which implies, when appropriate, setting the last trailing one to 0 or adding a one-value coefficient after the last nonzero coefficient. This method has low computationally complexity and, hence, can be

real-time realized. Experiments showed that the degradation of video quality is negligible and the same overall size of the video bit-stream is maintained.

Xu et al. [45] presented a scheme for data hiding directly in the encrypted H.264/AVC video bitstreams. The codewords of three sensitive parts (intraprediction modes, motion vector differences and levels) are encrypted with stream ciphers. Then, additional data may be embedded in the encrypted domain (specifically in levels codewords suffixes whose length is greater than one) without knowing the original video content. Data extraction can be done either in the encrypted domain or in the decrypted domain. In addition, experimental results showed that the file size is preserved and that the degradation in video quality caused by data hiding is quite small. In [46], Xu et al. proposed an improved version of their scheme that can achieve higher embedding capacity. Specifically, when the level suffix length is equal to 1, data embedding is performed by paired code-word substitution; when the level suffix length is greater than 2, the multiple-based notational system is adopted.

In addition to its use in video coding, CAVLC has many interesting applications and great possibilities in other areas of video and image compression, like medical image compression [20, 30, 37].

Sridhar et al. [37] proposed an advanced medical image compression technique based on integer DCT (Digital Cosine Transform), SPIHT (Set Partitioning In Hierarchical Trees) and CAVLC. Simulations on different medical images (including CT skull, angiogram and MR images) showed better results compared to JPEG and JPEG2000 schemes.

Mohanty et al. [20] presented a framework to stream histopathology image of a patient over a lossy network. Firstly, the image is divided into a number of fixed size tiles to provide access to regions of interest to the remote pathologist. Secondly, each tile is compressed using a proposed variant of WebP. Finally, a proposed greedy scheme packs macroblocks in such a way that that the number of undecodable received macroblocks is minimized. Although JPEG and JPEG2000 have been used to compress histopathology images, the authors selected WebP because the size of a file compressed by the former methods is 25%-34% more than that of the same file compressed by the last method [9, 10]. Nevertheless, they observed that the FCFS (First Come First Serve) inter-macroblock dependency introduced by WebP is not suitable to stream histopathology images because it cannot prioritize the decoding of an important macroblock. Hence, they modified WebP by using CAVLC in place of CABAC encoder.

Priya et al. [30] proposed a region-based compression method for compressing medical images in DICOM (Digital Imaging and Communications in Medicine) format. Their method consists of the following steps: (1) Using fuzzy C-means clustering, the image is segmented in regions of interest (ROI) and non-regions of interest (NROI). (2) The NROI and ROI areas are compressed using, respectively, CAVLC and a lossless compression method based on DWT (Discrete Wavelet Transform) and SPIHT. (3) The outputs of CAVLC and the lossless compression method are merged to get the compressed image. Experiments results showed that the presented method outperforms, in terms of PSNR (peak signal-to-noise ratio), SSIM (structural similarity index measure) and CR (compression ratio), the conventional

methods EZW (Embedded Zerotrees of Wavelet), STW (Spatial-orientation Tree Wavelet) and SPIHT.

7 Conclusions

This work has presented CAVLCU, a highly optimized GPU-based approach to CAVLC implemented in CUDA, which improves the only state-of-the-art implementation on GPU.

Thus, our algorithm outperforms the throughput of previous implementation by applying several optimization strategies. On the one hand, CAVLCU is built using only one kernel to avoid the long latency global memory accesses required to transmit intermediate results among different kernels, and the costly launches and terminations of additional kernels. On the other hand, our algorithm applies thread-block synchronization mechanism to manage efficiently the data dependence between thread-blocks in the calculation of the parameters nC . Moreover, CAVLCU optimizes the zigzag sorting of the blocks, as, after their reading through vectorized loads, sort them efficiently in the register space through few high throughput operations with high degree of instruction-level parallelism.

Experimental evaluation showed that CAVLCU is between 2.5 and 5.4 faster than the unique state-of-the-art GPU-based implementation.

We believe that our work is very useful for the following reasons. First, our algorithm is a significantly improved alternative to the only existing GPU-based solution. Second, our method can be exploited as the CAVLC component in GPU-based H.264 encoders, which are a very suitable solution when GPU built-in H.264 hardware encoders lack certain necessary functionality, such as data encryption and information hiding. Third, as CAVLC is a high-performance entropy compression method, apart from its wide use in the video standard H.264, it can be applied in many other compression systems. Hence, taking into account the massive use of multimedia data compression in the current digital era, our solution can be exploited in the development of many GPU-based applications for encoding both images and videos in formats other than H.264, like medical images. This is not possible with hardware implementations of CAVLC, as they are non-separable components of hardware H.264 encoders.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Babionitakis K, Doumenis G, Georgakarakos G, Lentaris G, Nakos K, Reisis D, Sifnaios I, Vlasopoulos N (2008) A real-time H. 264/AVC VLSI encoder architecture. *J Real-Time Image Process* 3(1–2):43–59
2. Banerji A, Ghosh AM (2010) *Multimedia Technologies*. Tata McGraw Hill, New Delhi
3. Chang C W, Lin W H, Yu H C, Fan CP (2014) A high throughput CAVLC architecture design with two-path parallel coefficients procedure for digital cinema 4K resolution H. 264/AVC encoding. In: *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on* (pp. 2616-2619). IEEE
4. Chu X, Wu S, Chang F, He W (2012) Efficient implementation of the CAVLC entropy encoder based on FPGA [J]. *J Xidian Univ* 3:017
5. Damak T, Werda I, Samet A, Masmoudi N (2008) DSP CAVLC implementation and optimization for H. 264/AVC baseline encoder. In: *Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on* (pp. 45-48). IEEE
6. El-Ghobashy WA, Ebian M, Mowafi O, Zekry AA (2015) An Efficient Implementation Method of H. 264 CAVLC video coding using FPGA. In: *Computer Engineering Conference (ICENCO), 2015 11th International* (pp. 212-216). IEEE
7. Fuentes-Alventosa A, Gómez-Luna J, González-Linares JM, & Guil N (2014) CUVLE: Variable-Length Encoding on CUDA. In: *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*(pp. 1-6). IEEE
8. Gómez-Luna J, Chang LW, Sung IJ, Hwu WM, Guil N (2015) In-Place Data Sliding Algorithms for Many-Core Architectures. In: *Parallel Processing (ICPP), 2015 44th International Conference on* (pp. 210-219). IEEE
9. Google Inc. WebP compression study. Draft 0.1 (May 2011). https://developers.google.com/speed/webp/docs/webp_study
10. Google Inc. Comparative study of WebP, JPEG, and JPEG2000 (August 2012). https://developers.google.com/speed/webp/docs/c_study
11. Hoffman MP, Balster EJ, Scarpino F, Hill K (2011) An Efficient Software Implementation of the CAVLC Encoder for H.264/AVC. In: *Proceedings of the 2011 IEEE National Aerospace and Electronics Conference (NAECON), Dayton, OH, , pp. 333-337*
12. Hoffman MP, Balster EJ, Turri WF (2016) High-throughput CAVLC architecture for real-time H. 264 coding using reconfigurable devices. *J Real-Time Image Process* 11(1):75–82
13. Hsia SC, Liao WH (2010) Forward computations for context-adaptive variable-length coding design. *IEEE Trans Circ Syst II Exp Briefs* 57(8):637–641
14. ITU-T Recommendation H.264 (2019) Advanced video coding for generic audiovisual services
15. Khronos group: OpenCL (2020). <https://www.khronos.org/opencv/>
16. Kim SM, Kim SB, Hong Y, Won CS (2007) Data Hiding on H. 264/AVC Compressed Video. In: *International Conference Image Analysis and Recognition* (pp. 698-707). Springer, Berlin, Heidelberg
17. Liao K, Lian S, Guo Z, Wang J (2012) Efficient information hiding in H 264/AVC video coding. *Telecommun Syst* 49(2):261–269
18. Luitjens J (2013) CUDA Pro Tip: Increase Performance with Vectorized Memory Access, Dec. <https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/>
19. Mian C, Jia J, Lei Y (2007) An H. 264 Video Encryption Algorithm Based on Entropy Coding. In: *Third International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP 2007)* (Vol. 2, pp. 41-44). IEEE
20. Mohanty M, Ooi W T (2012) Histopathology Image Streaming. In: *Pacific-Rim Conference on Multimedia* (pp. 534-545). Springer, Berlin, Heidelberg
21. Mukherjee R, Banerjee A, Maulik A, Chakrabarty I, Dutta PK, Ray AK (2017) An Efficient VLSI Design of CAVLC Encoder. In: *Region 10 Conference, TENCON 2017-2017 IEEE* (pp. 805-810). IEEE
22. NVIDIA: CUDA C Best Practices Guide 11.0 (2020). <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>
23. NVIDIA: CUDA Math API (2020) <https://docs.nvidia.com/cuda/cuda-math-api/index.html>
24. NVIDIA: CUDA Occupancy Calculator (2020) https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls

25. NVIDIA: CUDA C Programming Guide 11.0 (2020) <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
26. NVIDIA: CUDA Zone (2020) <https://developer.nvidia.com/category/zone/cuda-zone>
27. NVIDIA: NVENC Video Encoder API Programming Guide (2020) <https://docs.nvidia.com/video-technologies/video-codec-sdk/nvenc-video-encoder-api-prog-guide/index.html>
28. Orlandic M, Svarstad K (2017) An efficient hardware architecture of CAVLC encoder based on stream processing. *Microelectron J* 67:43–49
29. Ozer J (2016) *Encoding for Multiple Screen Delivery*. Udemy
30. Priya C, Ramya C (2018) Medical image compression based on fuzzy segmentation. *Int J Pure Appl Math* 118(20):603–610
31. Ren J, He Y, Wu W, Wen M, Wu N, Zhang C (2009) Software parallel CAVLC encoder based on stream processing. In: *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on* (pp. 126–133). IEEE
32. Richardson, Iain EG (2010) *The H.264 Advanced Video Compression Standard*, Wiley: Hoboken
33. Salomon D, Motta G (2010) *Handbook of data compression*. Springer, New York
34. Shahid Z, Chaumont M, Puech W (2009) Fast Protection of H. 264/AVC by Selective Encryption of CABAC. In: *2009 IEEE International Conference on Multimedia and Expo* (pp. 1038–1041). IEEE
35. Shahid Z, Chaumont M, Puech W (2009) Fast protection of H. 264/AVC by selective encryption. In: *Proceedings Of The Singaporean-French Ipal Symposium 2009: SinFra'09* (pp. 11–21)
36. Shahid Z, Chaumont M, Puech W (2011) Fast protection of H 264/AVC by selective encryption of CAVLC and CABAC for I and P frames. *IEEE Trans Circ Syst Video Technol* 21(5):565–576
37. Sridhar KV, Prasad KK (2008) Medical Image Compression Using Advanced Coding Technique. In: *2008 9th International Conference on Signal Processing* (pp. 2142–2145). IEEE
38. Su H, Wen M, Wu N, Ren J, Zhang C (2014) Efficient parallel video processing techniques on GPU: from framework to implementation. *The Sci World J*
39. Su H, Zhang C, Chai J, Wen M, Wu N, Ren J, A High-Efficient Software Parallel CAVCL Encoder Based on GPU. In: *2011 34th International Conference on Telecommunications and Signal Processing (TSP), Budapest, 2011*, pp. 534–540
40. Tabash FK, Izharuddin M (2017) Efficient Encryption Technique for H. 264/AVC Based on CAVLC and Baker's Map. In: *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)* (pp. 2759–2764). IEEE
41. Tabash FK, Izharuddin M, Tabash MI (2019) Encryption techniques for H. 264/AVC videos: a literature review. *J Inform Secur Appl* 45:20–34
42. Wang Y, O'Neill M, Kurugollu F (2013) A tunable encryption scheme and analysis of fast selective encryption for CAVLC and CABAC in H. 264/AVC. *IEEE Trans Circ Syst Video Technol* 23(9):1476–1490
43. Xiao Z, Baas B (2008) A High-Performance Parallel CAVLC Encoder on a Fine-Grained Many-Core System. In: *Computer Design. ICCD 2008. In: IEEE International Conference on* (pp. 248–254). IEEE
44. Xiph.org Video Test Media [derf's collection] (2020). <https://media.xiph.org/video/derf/>
45. Xu D, Wang R, Shi YQ (2014) Data hiding in encrypted H. 264/AVC video streams by codeword substitution. *IEEE Trans Inform Foren Secur* 9(4):596–606
46. Xu D, Wang R, Shi YQ (2016) An improved scheme for data hiding in encrypted H. 264/AVC videos. *J Vis Commun Image Rep* 36:229–242
47. Yan S, Long G, Zhang Y (2013) StreamScan: fast scan algorithms for GPUs without global barrier synchronization. *ACM Sigplan Notices* 48(8):229–238

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Antonio Fuentes-Alventosa¹ · Juan Gómez-Luna² ·
José Maria González-Linares³ · Nicolás Guil³ · R. Medina-Carnicer¹

✉ Antonio Fuentes-Alventosa
antonio.fa@gmail.com

Juan Gómez-Luna
juang@ethz.ch

José Maria González-Linares
jgl@uma.es

Nicolás Guil
nguil@uma.es

R. Medina-Carnicer
rmedina@uco.es

¹ Department of Computer Sciences and Numerical Analysis, University of Córdoba, Córdoba, Spain

² Department of Information Technology and Electrical Engineering, ETH Zürich, Zürich, Switzerland

³ Department of Computer Architecture, University of Málaga, Málaga, Spain

2.2. Segunda contribución: "GUD-Canny: a real-time GPU-based unsupervised and distributed Canny edge detector"

El segundo trabajo de esta tesis [36] propone GUD-Canny, una implementación en GPU no supervisada y distribuida del detector de bordes de Canny, que resuelve las dos principales limitaciones de sus implementaciones actuales [18, 37, 38, 39, 40, 41, 42, 43]:

- El cuello de botella causado por el proceso de histéresis.
- El uso de umbrales de histéresis fijos.

Dada una imagen $W \times H$, GUD-Canny calcula la magnitud del gradiente normalizado, la divide en sub-imágenes $32 \times H$ y calcula el par óptimo de umbrales de histéresis de cada sub-imagen utilizando el método de Medina-Carnicer [44]. Posteriormente, en lugar de ejecutar un costoso proceso de histéresis CPU-GPU de varias pasadas en toda la imagen, lleva a cabo un conjunto de procesos de histéresis (uno por sub-imagen, utilizando sus umbrales específicos) completamente en la GPU, de forma independiente y en paralelo. Cada bloque de hilos realiza el proceso de histéresis en una sub-imagen en memoria compartida, y representa cada píxel del mapa de histéresis con un solo bit para optimizar el uso del espacio limitado de dicha memoria.

La evaluación experimental mostró que GUD-Canny sólo requiere 0.35 ms en promedio para detectar los bordes de imágenes 512x512, por lo que satisface completamente el requisito de tiempo real, y es más rápido que las implementaciones en GPU [18, 37, 38, 39, 40, 41, 42, 43] y en FPGA [45, 46, 47, 48, 49, 50, 51, 52, 53] existentes.

Esta contribución se corresponde, como resultado, con el objetivo 3.



GUD-Canny: a real-time GPU-based unsupervised and distributed Canny edge detector

Antonio Fuentes-Alventosa¹ · Juan Gómez-Luna² · R. Medina-Carnicer¹

Received: 15 September 2021 / Accepted: 13 February 2022 / Published online: 5 March 2022
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2022

Abstract

The Canny algorithm is one of the most commonly used edge detectors due to its superior performance, especially in noisy environments. Its main limitation is that it is time consuming due to its multistage nature and the use of complex computational operations, primarily hysteresis thresholding. For this reason, many efficient implementations of the Canny edge detector have been developed on different accelerating platforms, such as ASICs, FPGAs and GPUs. The two main limitations of the GPU implementations developed to date are the bottleneck caused by the hysteresis process, and the use of fixed hysteresis thresholds. To overcome these issues, a novel GPU-based unsupervised and distributed Canny edge detector is proposed in this paper. Experimental evaluation showed that our Canny edge detector fully satisfies real time requirements, as it only requires 0.35 ms on average to detect edges on 512×512 images, and that it is faster than existing GPU and FPGA implementations.

Keywords Edge detection · Canny edge detector · GPU · CUDA · Parallel implementations

1 Introduction

Edge detection is an essential operation in different fields, such as image processing, computer vision and pattern recognition. Over the years, many edge detection algorithms have been proposed, including classical approaches, such as Roberts [1], Sobel [2], Prewitt [3] and Canny [4] methods, as well as more recent methods based on soft computing techniques, such as fuzzy logic [5], Artificial Neural Networks [6], genetic algorithms [7], particle swarm optimization [8], ant colony optimization [9] and adaptive neuro fuzzy inference system [10].

The Canny algorithm [4], also known as optimal detection method, is still one of the most widely used edge

detection techniques due to its superior performance. It consists of the following four stages: (1) noise reduction, (2) gradient computation, (3) non-maximum suppression, and (4) hysteresis thresholding. First, the image noise is reduced by a Gaussian convolution. Next, first derivatives are calculated in both horizontal (d_x) and vertical dimensions (d_y). From these two images, the gradient magnitude (G) and direction (θ) are computed for each pixel by the formulas $G = \sqrt{d_x^2 + d_y^2}$ and $\theta = \tan^{-1}(\frac{d_y}{d_x})$. In the third stage, possible edges are obtained by suppressing all pixels which are not local maximums in the gradient direction. In the last stage, hysteresis thresholding determines which of possible edges are really edges using two thresholds values, *low* and *high*. First, the set of pixels with $G \geq high$ and the set of pixels with $G \leq low$ are directly classified as edges and non-edges, respectively. Then, the remaining possible edges (i.e., those with $low < G < high$) are classified as edges if and only if they are connected (directly or via other possible edges) to pixels with $G \geq high$. In the rest of the paper, the set of pixels with $low < G < high$ will be referred to as *instability zone* [11], and their classification process as *linking process* [11]. Additionally, we define the *instability map* as a binary image of the same dimensions as G , in which the value of pixel (i, j) is 1 if the pixel (i, j) of G belongs to the instability zone, or 0 otherwise.

✉ Antonio Fuentes-Alventosa
antonio.fa@gmail.com

Juan Gómez-Luna
juang@ethz.ch

R. Medina-Carnicer
rmedina@uco.es

¹ Department of Computer Sciences and Numerical Analysis, University of Córdoba, Córdoba, Spain

² Department of Information Technology and Electrical Engineering, ETH Zürich, Zürich, Switzerland

The main drawback of the Canny edge detector is that it is time consuming, due to its high computational complexity. To overcome this limitation, many implementations of the algorithm have been presented on different accelerating platforms, such as ASICs [12–14], FPGAs [15–23] and GPUs [25–32].

There are several ASICs implementations of Deriche filters, which have been derived from Canny's criteria. Deriche [12] presented a network with four transputers that took 6 s to detect edges in a 256×256 image, which is far from real-time requirements. Torres et al. [13] proposed a faster solution that processed 25 frames/s at 33 MHz, but the area overhead was increased by the use of Last-In First-Out (LIFO) stacks in off-chip SRAM memories. Lorca et al. [14] presented a new design that improved that of [13] by reducing the memory size and the computation cost by a factor of two. Nevertheless, the number of clock cycles per pixel varies with the image size, and the processing time increases with the size of the image.

Some efforts have been made to accelerate Canny edge detection using FPGAs [15–23]. The proposals in [15] and [16] translated the software designs directly into hardware description languages (Handel-C and VHDL, respectively), which resulted in timing performance degradation. Gentsos et al. [17] presented a parallel architecture of simultaneous 4-pixel calculation that reduced the latency of the implementations of [15] and [16]. He et al. [18] proposed a self-adapt threshold Canny algorithm to overcome the drawback of setting the hysteresis thresholds manually in existing hardware implementations. In their method, hysteresis thresholds are calculated from the histogram of gradient magnitude. Their algorithm required about 2.5 ms to detect the edges of a 360×280 image on a FPGA chip EP1C60240C8 (Altera Cyclone) based platform. Li et al. [19] presented other solution for self-adapt threshold Canny algorithm, which adopted a Shifting-LUT-based direction calculation algorithm to improve the processing speed. The processing time was 5.24 ms for a 512×512 image on a Xilinx's Virtex-5 FPGA. Peng et al. [20] proposed an improved high-speed Canny edge detection algorithm based on FPGA, in which the gradient is calculated by the second harmonic of the variable parameters (SHOVP) to simplify complex arithmetic into logic operation. The feasibility and effectiveness of the algorithm was tested on Altera DE2 platform. Abdelgawad et al. [21] proposed an implementation of Canny algorithm on Zynq platform using Vivado High Level Synthesis (HLS). The achieved results showed that the collaboration of CPU and FPGAs enabled up to a 100x performance improvement. The CPU utilization dropped down and the frame rate was up to 60 fps for 1280×1024 resolution. Xu et al. [22] presented a distributed Canny edge detection algorithm that adaptively computes the edge detection thresholds based on the block type and the local distribution of the gradients in the image

block. In addition, their method uses a non-uniform gradient magnitude histogram to compute block-based hysteresis thresholds. The implementation of the algorithm on a Xilinx Virtex-5 FPGA platform takes only 0.721 ms (including the SRAM read/write time and the computation time) to detect edges of 512×512 images in the USC SIPI database when clocked at 100 MHz. Sangeetha et al. [23] proposed a cost-effective robust Canny edge detection algorithm, whose keys contributions are the following: (1) computation of gradient magnitude and orientation using approximate method, (2) block classification techniques, and (3) adaptive threshold calculation of each block. Results on Xilinx Virtex-5 FPGA showed that the algorithm requires only 0.672 ms to detect the edges of 512×512 image when clocked at 100 MHz.

In the area of General Purpose Graphic Processing Unit (GPGPU), several efficient implementations of the Canny algorithm have been proposed [25–32]. Luo and Duraiswami [25] presented the first implementation of the Canny algorithm on the popular NVIDIA CUDA framework [33]. They mapped the entire algorithm to the GPU, and improved previous similar implementations on NVIDIA Cg [34] and Khronos Group GLSLang [24] that not included the hysteresis stage. The convolution steps (Gauss and Sobel filtering) are efficiently implemented using a separable filter algorithm, similar to the one supplied with the CUDA toolkit [35]. The gradient magnitude and direction are easily obtained by calculating the L2 norm and the arctangent, respectively, of the first derivatives on a simple pixel to thread mapping. The gradient direction of each pixel is quantized to one of the eight directions corresponding to the neighboring pixels ($\pi/8 + k\pi/4$). Non-maximum suppression is performed on a straightforward way by setting to 0 the gradient magnitudes that are not local maximums in the gradient direction. Hysteresis is performed by a kernel of 16×16 thread-blocks, each of which processes a separate 16×16 pixel-block of the gradient along with a one pixel wide apron around the 16×16 pixel-block, resulting in a 17×17 pixel-block. Each thread-block loads its assigned 17×17 pixel-block to shared memory, and executes a breadth first search (BFS) algorithm on it to classify the pixels of the internal 16×16 pixel-block as edges or non-edges. This classification is carried out by assigning -2 to the gradient magnitude, if the pixel is an edge, or 0, otherwise. Once a thread-block finishes the BFS process, it writes the edge states of all non-apron pixels in shared memory back into the gradient magnitude space in global memory. Subsequent calls to the hysteresis kernel will allow the linking among pixels that belong to different 16×16 pixel-blocks, thanks to the reloading of the updated edge states of apron pixels into shared memory. Due to this multi-pass approach, the implementation speed is dominated by the hysteresis process. Experimental evaluation showed that it occupies more than 70% of the total runtime. For testing purposes, the

hysteresis kernel was called four times per iteration, as no significant improvement was observed with higher values for the test images. Experiments showed a significant speedup against straightforward CPU functions, but a moderate improvement against multi-core multi-threaded CPU functions taking advantage of special instructions. The measured execution time for a 512×512 image was 3.40 ms. Ogawa et al. [26] presented a solution based on the work of Luo and Duraiswami [25], in which they described an issue in the traversing of all weak edge pixels, and proposed a stack-based mechanism to fix it. In the hysteresis thresholding stage, if the pixel assigned to a thread is a strong edge pixel, the thread uses a stack to traverse the adjacent weak edge pixels, which are labeled as final edge pixels. Experimental evaluation showed a runtime of 364.389 ms for a 10240×10240 image. The logarithmic image processing (LIP) model is a robust mathematical framework that is compatible with what is known about the human visual process [36]. In [27], Palomar et al. presented the implementation of two LIP-Canny methods, one operating images in LIP space with traditional operators, and the other operating images in natural space with modified operators. The work of Palomar et al. [27] was based on those of Palomares et al. [37] and Luo and Duraiswami [25]. As in [25], the number of iterations of the hysteresis kernel was fixed to 4. Experimental evaluation showed that CUDA implementations are 10–16 times faster than the corresponding C++ implementations. Moreover, LIP-Canny using modified operators is slightly faster than the alternative approach based on classical operators. The average runtimes for 512×512 images were 26.448 ms and 28.848 ms for the first and second method, respectively. Lourenço et al. [28] developed a CUDA implementation of the Canny algorithm for the Insight Segmentation and Registration Toolkit (ITK) using second-order derivatives (instead of Sobel filtering [25]) and a hybrid CPU-GPU approach for the hysteresis stage that closely followed the method proposed in [25]. Experimental evaluation showed that the CUDA implementation on three generations of NVIDIA GPGPUs was between 3.6 and 50 times more faster than the standard ITK Canny implementation on two CPU models. The main novelties of the CUDA implementation proposed by B. M. L. P. Vigil [29] are the application of Otsu method for automatic calculation of hysteresis thresholds, and the use of interpolation in the non-maximum suppression step to improve the quality of edge detection. The hysteresis thresholding is performed by the same hybrid CPU-GPU technique used in previous works, and, hence, it occupies a considerable percentage in the total execution time (more than 50%). The execution times of the CUDA Canny detector for 512×512 Lena, Mandrill and Peppers images were 8.49 ms, 9.84 ms and 10.90 ms, respectively. Huang et al. [30] presented a CUDA implementation on the embedded CPU and GPU heterogeneous computing platform Jetson TK1 of NVIDIA.

Noise reduction, gradient computation and non-maximum suppression are efficiently implemented in a similar way to that of [25]. However, the linking process is replaced by a simpler schema, which classifies a pixel of the instability zone as an edge pixel if at least one of its eight neighboring pixels is an edge pixel. Additionally, the hysteresis thresholds are obtained from the histogram of gradient magnitude. Experimental evaluation showed that the runtimes for 512×512 Lena and Peppers images were approximately 3 ms. In [32], Emrani et al. presented a CUDA implementation of Canny algorithm in which the main novelty was the replacement of the Luo and Duraiswami's BFS algorithm [25] with a more efficient method. The kernel corresponding to this method checks whether a pixel belongs to the instability zone or not. If so, it will check its neighboring pixels. If a strong edge is found, the current pixel is classified as an edge pixel. A flag in global memory is used to indicate whether any pixel of the instability zone has been classified as an edge pixel. The kernel is launched as long as the flag is set. The execution time of the CUDA Canny detector for a 512×512 image was 37.35 ms on a GeForce GTX 550 Ti GPU.

As we have just seen, the main bottleneck of GPU-based implementations of Canny algorithm is the hysteresis step, due to the need of calling the hysteresis kernel an indeterminate number of times (at least 4) executed on host side. On the other hand, in all implementations, except B. M. L. P. Vigil's [29] and Huang et al.'s [30], the hysteresis thresholds are adjusted manually. In this work, we propose a novel GPU-based implementation of the Canny algorithm on CUDA that overcomes these limitations. As in [22] and [23], the image is partitioned into sub-images, and the following steps are performed on each sub-image in parallel: (1) calculation of the optimal hysteresis thresholds, and (2) hysteresis process using the parameters obtained in the previous step. As each sub-image is processed independently, it is not necessary the costly hybrid CPU-GPU approach of previous implementations for hysteresis stage. The calculation of hysteresis thresholds is carried out with Medina-Carnicer's method [11], which, at present, is relevant for unsupervised edge detection because, since its introduction, it has been used to find automatically the hysteresis thresholds in many works [38–51]. Medina-Carnicer's method [11] outperforms those used in previous implementations of Canny algorithm [18, 19, 22, 23, 29], because the first searches the optimal values of both hysteresis thresholds *low* and *high*, while the latter do not, since they assume a constant ratio *low/high*. Experimental evaluation showed that our GPU-based unsupervised and distributed Canny edge detector, which we have named GUD-Canny, requires only between 0.33 and 0.48 milliseconds to detect edges on 512×512 images, which fully satisfies real-time requirements and outperforms reported runtimes of existing FPGA and GPU solutions.

The rest of the paper is organized as follows. Section 2 gives a brief overview of Medina-Carnicer’s method. Section 3 presents GUD-Canny. Section 4 shows the experimental evaluation of our solution, and, finally, the main conclusions are stated in Sect. 5.

2 Medina-Carnicer’s method for unsupervised determination of hysteresis thresholds

2.1 Background

In [11], Medina-Carnicer et al. presented a novel method to look for the hysteresis thresholds in an unsupervised way. Given a set of candidate thresholds pairs, the key idea is to combine the gradient information with that obtained from applying the linking process for all the candidate thresholds pairs. Experimental evaluation showed that the performance of Medina-Carnicer’s algorithm is better than those of previous methods [52, 53]. The computational complexity of Medina-Carnicer’s algorithm [11] is smaller than that of the solution presented in [53], but bigger than that of the proposal in [52]. Nevertheless, the approach in [52] only finds an approximate edge map and it is not able to find the hysteresis thresholds. The results obtained by Medina-Carnicer’s method [11] have been validated only for the Canny edge detector, but there are no restrictions to apply it to any other edge detector whose strategy is based on the hysteresis mechanism.

The main innovations presented in [11] are the following:

1. In contrast to previous works [53–56], which are aimed at directly searching for hysteresis thresholds, it follows an indirect way, which consists of looking for the instability zone and then determining the hysteresis thresholds from it.
2. Unlike previous proposals [53, 55, 56], which only use gradient information, it combines the latter with that of the linking process.

2.2 Steps summary of Medina-Carnicer’s method

Let I be an image, G its gradient magnitude after non-maximum suppression normalized in the interval $[0,1]$, and C a set of candidate thresholds pairs $\{(low, high), low, high \in (0, 1)\}$.

Given a hysteresis thresholds pair $(low, high)$, we define the following edge maps:

- *Hysteresis map* ($G_{low,high}$), which is obtained by performing the hysteresis process on G with $(low, high)$.
- *High map* (G_{high}), which is the result of thresholding G with $high$.

- *Linking map* ($\Delta G_{low,high}$), which is composed exclusively of the edges added by the linking process using $(low, high)$. Note that $\Delta G_{low,high} = G_{low,high} - G_{high}$.

The steps of Medina-Carnicer’s method are the following:

1. Calculate a set H of linking maps corresponding to the candidate thresholds pairs of C .

$$H = \{\Delta G_{low,high}, (low, high) \in C\} \tag{1}$$

2. Compute the sum SM_H of the linking maps.

$$SM_H = \sum H \tag{2}$$

In this matrix, the value of each element is the number of times that the corresponding pixel of G is classified as edge by the linking process for all the candidate thresholds pairs.

3. Calculate the division of SM_H by the cardinality of C , which will be denoted as $Prob(SM_H)$.

$$Prob(SM_H) = SM_H/|C| \tag{3}$$

Each element of $Prob(SM_H)$ represents the probability that the corresponding pixel of G is classified as edge by the linking process.

4. Compute the distribution $P(F(x)), \forall x \in (0, 1)$, defined as follows:

$$P(F(x)) = \begin{cases} \frac{|F(x)|}{|Prob_x(SM_H)|} & |Prob_x(SM_H)| > 0 \\ 0 & |Prob_x(SM_H)| = 0 \end{cases} \tag{4}$$

where

- $Prob_x(SM_H)$ is the binary edge map obtained by thresholding $Prob(SM_H)$ with $x \in (0, 1)$. Its elements with value 1 correspond to the pixels of G that have a probability equal or greater than x of being classified as edges by the linking process.
- $|Prob_x(SM_H)|$ is the number of elements with value 1 in $Prob_x(SM_H)$.
- $F(x) = G \circ Prob_x(SM_H)$, where \circ is the Hadamard product.
- $|F(x)|$ is the number of elements with value x in $F(x)$.

The distribution $P(F(x))$ represents the probability that a pixel has gradient level x if it is a pixel with probability equal or greater than x of being added by the linking process. It is the combined information used by Medina-Carnicer’s method.

5. Compute the histogram of $Prob(SM_H)$ for the set $D = \{x \in (0, 1) | P(F(x)) \neq 0\}$, which represents the instability zone. The hysteresis thresholds are the values of D

corresponding to the first and last local maximums of the histogram.

The set C is obtained by sampling an interval $[0.01, MAX_HIGH]$, where $0.01 < MAX_HIGH \leq 1.0$. In [11], Medina-Carnicer et al. showed that two selections of C that ensure a good performance of their method are those obtained by sampling the interval $[0.01, 0.25]$ with steps 0.01 and 0.03. Furthermore, the results presented in [53] indicate that their approach, in general, depends less on the initial set than the method of Yitzhaky and Peli [56] does.

3 GPU-based unsupervised and distributed Canny edge detector (GUD-Canny)

In this section, we describe *GUD-Canny*, our GPU-based unsupervised and distributed Canny edge detector, which has been developed using the popular NVIDIA CUDA framework [33]. In the presented algorithms, the following notation is employed:

- Prefixes $d_$, $s_$ and $c_$ in the names of the variables indicate that they are allocated in global, shared and constant memory spaces, respectively.
- Symbols $\&$, $|$, \sim , \ll and \gg are the bitwise operators AND, OR, NOT, left shift and right shift, respectively.

Algorithm 1 provides a high-level description of GUD-Canny. As it can be seen, the inputs of our method are the following. First, a $W \times H$ image, which is provided in a vector of P 8-bit unsigned integers (d_image), where P is the number of pixels. Second, the standard deviation σ . Third, a set of $NCTP$ candidate thresholds pairs, which is supplied in a vector of float pairs (c_C). On the other hand, the output of GUD-Canny are the edges of the input image, which are written in a vector of P 8-bit unsigned integers (d_edges).

Steps 1–3 correspond to the classic first stages of Canny edge detection. To apply Medina-Carnicer's method (steps 4 to 7), the non-maximum suppression returns the gradient magnitude normalized in the interval $[0, 1]$ (d_G). The gradient magnitude is partitioned horizontally into $NS = W/32$ sub-images of dimension $32 \times H$, and Medina-Carnicer's method [11] is used to calculate an optimal pair of hysteresis thresholds for each sub-image. Finally, in step 8, the hysteresis map is computed for each subimage using its assigned hysteresis thresholds pair, and written in the output vector d_edges .

Since the original width of the input image may not be a multiple of 32, the CUDA function *cudaMemcpy2D* [57] is

used to copy the input image from host to device memory adding the necessary padding to each row, and the same function is called to copy the output edges from device to host memory.

In the following subsections, each step of GUD-Canny is described in detail.

Algorithm 1: GUD-Canny

```
function GUD-Canny(output
    uchar d_edges[P],
    // input
    uchar d_image[P],
    float  $\sigma$ ,
    float2 c_C[NCTP]
)
// Step 1: Gaussian filtering
uchar d_smoothed_image[P]
d_smoothed_image  $\leftarrow$  gaussian_filter(d_image,  $\sigma$ )

// Step 2: gradient computation
short d_grad_x[P], d_grad_y[P],
    d_grad_mag[P], d_max_grad_mag[1]
(d_grad_x, d_grad_y, d_grad_mag,
 d_max_grad_mag)  $\leftarrow$  compute_gradient(d_smoothed_image)

// Step 3: non-maximum suppression
float d_G[P]
d_G  $\leftarrow$  non_maximum_suppression(d_grad_x, d_grad_y,
    d_grad_mag, d_max_grad_mag)

// Step 4: calculation of matrices  $SM_H$ 
uint d_SM_H[P]
d_SM_H  $\leftarrow$  calc_SM_H(d_G, c_C)

// Step 5: calculation of matrices Prob( $SM_H$ )
float d_Prob_SM_H[P]
d_Prob_SM_H  $\leftarrow$  calc_Prob_SM_H(d_SM_H, NCTP)

// Step 6: calculation of distributions  $P(F(x))$  and
// histograms of matrices Prob( $SM_H$ )
float d_PF[NS][NX + 1]
uint d_histo[NS][NX + 1]
(d_PF, d_histo)  $\leftarrow$  calc_PF_histo(d_G, d_Prob_SM_H)

// Step 7: searching of hysteresis thresholds for
// each subimage of G
float2 d_thresholds[NS]
d_thresholds  $\leftarrow$  search_thresholds(d_PF, d_histo)

// Step 8: hysteresis thresholding
d_edges  $\leftarrow$  hysteresis(d_G, d_thresholds)
end function
```

3.1 Gaussian filtering

To reduce the impact of noise, the input image is smoothed by convolving it with two one-dimensional Gaussian filters in the horizontal and vertical dimensions.

Each Gaussian filtering is performed by a different CUDA kernel, in which each output pixel is computed by a different thread. Kernels implementations are similar to those

presented in [35], but with the difference that the shared memory is not used for caching data. Since the hardware cache system ensures a good performance [57], all read/write operations are performed directly to global memory.

Each thread initializes each element of the input image vector used to perform the convolution dot product as follows. If it corresponds to an existing pixel, i.e., the position of the pixel is not outside the borders of the image, it is read from the input image. Otherwise, it is assigned the value zero.

As in [27], the length of Gaussian filters is variable and depends on the standard deviation σ . Each kernel obtains the Gaussian filter from a table in constant memory, which stores the Gaussian filters corresponding to σ values between 0.1 and 2.0. The first table entry corresponds to $\sigma = 0.1$, the second one to $\sigma = 0.2$, and so on up to 2.0.

3.2 Gradient computation

After Gaussian filtering, each gradient tuple (d_x, d_y) is calculated using the first difference operator $(-1, 0, 1)$, and the associated gradient magnitude by the formula $\sqrt{d_x^2 + d_y^2}$. The results are written in the output vectors d_grad_x , d_grad_y and d_grad_mag , respectively. As in Gaussian filtering step, all read/write operations are made directly to global memory, and the border conditions are carefully checked.

Additionally, to compute the maximum gradient magnitude, each thread performs an atomic maximum operation (using the CUDA function *atomicMax* [57]) between the calculated gradient magnitude and a global memory variable ($d_max_grad_mag[0]$), which has been initialized to zero.

3.3 Non-maximum suppression

In this step, one kernel computes a new version of gradient magnitude (d_G) by performing non-maximum suppression and normalization on the gradient magnitude obtained in the previous stage (d_grad_mag). Given a pixel of value p in d_grad_mag , the value of the corresponding pixel in d_G is $p/d_max_grad_mag[0]$ if the pixel is a maximum in the gradient direction, or zero otherwise. Each pixel of d_G is computed by a different thread of the kernel.

The method used for maximum suppression is the one employed in [29], which quantizes gradient direction to one

of the eight directions $\{\pi/8 + k\pi/4\}$, and uses linear interpolation to calculate the values of the two neighboring pixels in the gradient direction.

Global memory operations and border conditions management are executed as in previous steps.

3.4 Hysteresis thresholds computation

As we said previously, the gradient magnitude is partitioned horizontally into $NS = W/32$ sub-images, and the method of Medina-Carnicer is applied on each one in parallel.

Images are processed by dividing them into groups of 32 consecutive pixels in the horizontal dimension, which will be referred to as *regions*. The numbers of regions of an image and of a sub-image will be denoted by *NRI* and *NRS*, respectively. For simplicity, the regions of instability/hysteresis/high/linking maps will be referred to as *instability/hysteresis/high/linking regions*, respectively.

Each of the steps 4–7 of Algorithm 1 is performed by a different kernel, whose actions are specified in the following subsections.

3.4.1 Calculation of the matrices SM_H

Algorithm 2 presents the pseudo code of the kernel *calc_SM_H*, which calculates the matrix SM_H for each sub-image of G . The inputs are G , which is provided in a vector of P 32-bit floats (d_G), and C , which is supplied in a vector of $NCTP$ 32-bit float pairs, initialized statically in constant memory (c_C). The output are the NS matrices SM_H corresponding to the NS sub-images of G , which are written in a vector of P 32-bit unsigned integers (d_SM_H), initialized to 0. Maps regions are represented by 32-bit unsigned integers, where the i -th bit stores the binary value of the i -th pixel of the region. Although the gradient regions reads in step 1 are not coalesced, as CUDA literature [58] [57] recommends, they satisfy the principle of spatial locality because each thread reads 32 consecutive elements of d_G , which are properly aligned. Therefore, the transparent cache hierarchy of modern GPU architectures ensures a good performance while reading the gradient regions. On the other hand, the writes in step 5 are carried out atomically using the CUDA function *atomicAdd* [57].

Algorithm 2: calc_SM_H

```

function calc_SM_H(// output
    uint d_SM_H[P],
    // input
    float d_G[P],
    float2 c_C[NCTP]
)
// Declarations
float low, high, grad_reg[32]
uint high_reg, inst_reg, hyst_reg, link_reg
uint *d_SM_H_reg

// Step 1: read input data
{low, high} ← read candidate thresholds pair assigned
    to current thread-block from c_C
grad_reg ← read gradient region assigned to current
    thread from d_G

// Step 2: calculate high and instability regions
high_reg ← 0
inst_reg ← 0
for i ← 0 to 31 do
    if (grad_reg[i] ≥ high) then
        set i-th bit of high_reg to 1
    else if (low < grad_reg[i] < high) then
        set i-th bit of inst_reg to 1
    end if
end for

// Step 3: calculate hysteresis region
hyst_reg ← calc_hyst_map(high_reg, inst_reg)

// Step 4: calculate linking region
link_reg ← hyst_reg & ~high_reg

// Step 5: update SMH region
d_SM_H_reg ← pointer to d_SM_H subvector corresponding
    to SMH region assigned to current thread
for i ← 0 to 31 do
    if (i-th bit of link_reg is 1) then
        atomicAdd(d_SM_H_reg[i], 1)
    end if
end for
end function

```

In step 3, of Algorithm 2 each thread gets its hysteresis region (*hyst_reg*) by calling the function *calc_hyst_map*, which receives as inputs the high and instability regions of the calling thread (*high_reg* and *inst_reg*, respectively). The actions performed by this function are presented in Algorithm 3. As it can be seen, each thread-block computes its hysteresis map in a shared memory 32-bit unsigned int vector (*s_hyst_map*) of size *NRS*. Each hysteresis region *i* is managed by the thread *i*, and held in the element *s_hyst_map*[*i*].

Algorithm 3: calc_hyst_map

```

function calc_hyst_map(// output
    uint hyst_reg,
    // input
    uint high_reg,
    uint inst_reg
)
// Declarations
shared uint s_hyst_map[NRS]
shared bool s_flag_updated
uint top_hyst_reg, bottom_hyst_reg
uint link_mask, new_hyst_edges

// Step 1: initialize the hysteresis map
// to the high map
s_hyst_map[thread_idx] ← high_reg
hyst_reg ← high_reg

repeat
    // Step 2: reset the flag s_flag_updated, which
    // is used to control the execution of the loop
    if thread_idx = 0 then
        s_flag_updated ← false
    end if
    synchronize all threads of the block

    // Step 3: check if the instability region
    // has edges
    if inst_reg ≠ 0 then
        // Step 4: read the top and bottom
        // hysteresis regions from the hysteresis map
        top_hyst_reg ← s_hyst_map[thread_idx - 1]
        bottom_hyst_reg ← s_hyst_map[thread_idx + 1]

        // Step 5: get the new edges to include
        // in the hysteresis region
        link_mask ← (hyst_reg << 1) & (hyst_reg >> 1)
            & top_reg & (top_reg << 1) & (top_reg >> 1)
            & bottom_reg & (bottom_reg << 1)
            & (bottom_reg >> 1)
        new_hyst_edges ← inst_reg & link_mask

        // Step 6: check if there are new edges
        if new_hyst_edges ≠ 0 then
            // Step 7: add the new edges to the
            // hysteresis region, and exclude them
            // from the instability region
            hyst_reg ← hyst_reg | new_hyst_edges
            inst_reg ← inst_reg & ~new_hyst_edges

            // Step 8: update the hysteresis map
            s_hyst_map[thread_idx] ← hyst_reg

            // Step 9: Register that the hysteresis map
            // has been updated
            s_flag_updated ← true
        end if
    end if

    // Step 10: thread-block synchronization
    synchronize all threads of the block

    // Step 11: if the hysteresis map has not been
    // updated in the last iteration, the loop ends
    until s_flag_updated = false
end function

```

An alternative way to divide the gradient magnitude into sub-images is by partitioning it vertically into $NS = H/32$ sub-images of dimension $W \times 32$. In this case, the spatial locality of accesses to global memory is improved, because consecutive threads access consecutive regions. On the other hand, the advantage of the horizontal partition is that the

number of operations in the linking process is reduced (step 5 of the function *calc_hyst_map*). The reason is that it is only necessary to examine the top and bottom regions; in the case of a vertical partition, the six remaining neighbor regions (left, top left, bottom left, right, top right and bottom right) have also to be taken into account. As will be shown in Sect. 4, GUD-Canny is slightly faster for sub-images of dimension $32 \times H$.

3.4.2 Calculation of the matrices $Prob(SM_H)$

The matrix $Prob(SM_H)$ for each sub-image of G is obtained by dividing each element of the corresponding matrix SM_H by $NCTP$. The matrices $Prob(SM_H)$ are written in a vector of P 32-bit floats ($d_Prob_SM_H$).

The number of threads of the grid equals to P divided by 4, and each thread i performs the following actions:

1. Reads the group i of four consecutive elements from d_SM_H through one vectorized load.
2. Calculates the division of each element by $NCTP$.
3. Writes the four computed float values to the 4-elements group i of $d_Prob_SM_H$ through one vectorized store.

Vectorized accesses are an important GPU optimization, because they increase bandwidth and reduce both instruction count and latency [59].

3.4.3 Calculation of the distributions $P(F(x))$ and the histograms of the matrices $Prob(SM_H)$

For each sub-image of G , the distribution $P(F(x))$ and the histogram of $Prob(SM_H)$ are computed by one kernel for $x \in \{0.01, 0.02, \dots, MAX_HIGH\}$. The number of x values, which is $MAX_HIGH/0.01$, will be denoted by NX .

The number of thread-blocks of the grid is $NS \times NX$. Each thread-block calculates $P(F(x))$ and the histogram of $Prob(SM_H)$ for one sub-image of G and one x value. The size of thread-blocks is NRS .

The actions performed by the kernel are shown in Algorithm 4, where *div* and *mod* are the quotient and remainder operators, respectively. The three parallel reductions are efficiently executed using the CUDA function `__shfl_down_sync` [57] and fast device memory atomic operations, as described in [60].

Algorithm 4: calc_PF_histo

```
function calc_PFI_histo(// output
    float d_PF[NS][NX + 1],
    uint d_histo[NS][NX + 1],
    // input
    float d_G[P],
    float d_Prob_SM_H[P]
)
// Declarations
uint x_idx, subimage_idx, region_idx
float x, Prob_reg[32], G_reg[32], dist_value
uint Prob_sum, total_Prob_sum, G_sum, total_G_sum,
    hist_sum, total_hist_sum

// Step 1: initializations
x_idx ← 1 + (thread_block_index mod NX)
subimage_idx ← thread_block_index div NX
region_idx ← thread_index
x ← 0.01 × x_idx

// Step 2: process region of Prob(SMH)
Prob_reg ← read region region_idx from sub-image
            subimage_idx of d_Prob_SM_H
Prob_sum ← count all elements of Prob_reg that are
            greater than or equal to x

// Step 3: process region of G
if (Prob_sum > 0) then
    G_reg ← read region region_idx from sub-image
            subimage_idx of G
    G_sum ← count all elements of G_reg that are equal
            to x whose associated elements in Prob_reg
            are greater than or equal to x
end if

// Step 4: thread-block reductions
total_Prob_sum ← thread-block reduction of Prob_sum
total_G_sum ← thread-block reduction of G_sum

// Step 5: the first thread of the thread-block
// calculates the P(F(x)) value and writes it
// to d_Prob_SM_H
if (thread_idx = 0) then
    if (total_Prob_sum = 0) then
        dist_value ← 0
    else
        dist_value ← total_G_sum / total_Prob_sum
    end if
    d_PF[subimage_idx][x_idx] ← dist_value
end if

// Step 6: check if the histogram value is non-zero
if (total_G_sum > 0) then
// Step 7: process region of Prob(SMH)
    hist_sum ← count all elements of Prob_reg that
                are equal to x
    total_hist_sum ← thread-block reduction of
                    hist_sum

// Step 8: the first thread of the thread-block
// writes the histogram value to d_histo
if (thread_idx = 0) then
    d_histo[subimage_idx][x_idx] ← total_hist_sum
end if
end if
end function
```

3.4.4 Searching of hysteresis thresholds

The hysteresis thresholds searching for each sub-image of G is performed by the kernel described in Algorithm 5. The number of warps of the grid is NS , and each warp i searches for the hysteresis thresholds pair of sub-image i . It is assumed that $NX < 32$.

The warp votes are performed by calling the CUDA function `__balloc_sync` [57], which, given a predicate, evaluates it for all threads in the current warp, and returns a 32-bit binary mask, in which each bit j is set if the predicate evaluates to non-zero for the lane j .

The searches of bits within the masks are performed efficiently using the CUDA integer intrinsic functions `__ffs` and `__brev` [61]. The first one finds the position of the least significant bit set to 1 in a 32-bit integer, and the second one reverses the bit order of a 32-bit unsigned integer.

Algorithm 5: search_thresholds

```
function search_thresholds(// output
    float2 d_thresholds[NS],
    // input
    float d_PF[NS][NX + 1],
    uint d_histo[NS][NX + 1]
)
// Declarations
uint curr_x_idx, subimage_idx, left_x_idx, right_x_idx,
    x_index_of_first_local_max, x_index_of_last_local_max
float curr_x, PF_x, hist_x, hist_left_x,
    hist_right_x, low, high
uint PF_mask, hist_mask
bool local_mask

// Step 1: initializations
curr_x_idx ← warp_lane
subimage_idx ← grid_warp_idx
curr_x ← 0.01 × curr_x_idx

// Step 2: each thread of the warp reads its assigned
// values in P(F(x)) and the histogram of Prob(SMi(x))
if curr_x_idx ≤ NX then
    PF_x ← d_PF[subimage_idx][curr_x_idx]
    hist_x ← d_histo[subimage_idx][curr_x_idx]
else
    PF_x ← 0
    hist_x ← 0
end if

// Step 3: each thread of the warp gets the indices
// of the x values for which P(F(x)) and the histogram
// of Prob(SMi(x)) are non-zero
PF_mask ← warp-voting(PF_x ≠ 0)
hist_mask ← warp-voting(hist_x ≠ 0)

// Step 4: determine the local maximums of the
// histogram of Prob(SMi(x)) for which P(F(x)) ≠ 0
if PF_x = 0 then
    local_max_mask ← false
else
    left_x_index ← find index of first one bit of
        PF_mask at right of bit of index curr_x_idx
    right_x_index ← find index of first one bit of
        PF_mask at left of bit of index curr_x_idx

    if left_x_index and right_x_index were found then
        hist_left_x ← d_histo[subimage_idx][left_x_idx]
        hist_right_x ← d_histo[subimage_idx][right_x_idx]
        local_max ← (hist_x > hist_left_x) and
            (hist_x > hist_right_x)
    else
        local_max ← false
    end if
end if
local_max_mask ← warp-voting(local_max = true)

// Step 5: search the first and last local maximums,
// which correspond to the low and high hysteresis
// thresholds, respectively
x_index_of_first_local_max ← find index of first one
    bit of local_max_mask
x_index_of_last_local_max ← find index of last one
    bit of local_max_mask

// Step 6: return the hysteresis thresholds pair
low ← 0.01 × x_index_of_first_local_max
high ← 0.01 × x_index_of_last_local_max
d_thresholds[subimage_idx] ← {low, high}
end function
```

3.5 Hysteresis thresholding

The hysteresis thresholding is carried out by the kernel presented in Algorithm 6, which is very similar to Algorithm 2. The number of thread-blocks of the grid is NS , and the size of each thread-block is NRS . The i -th thread-block calculates the hysteresis map corresponding to the i -th sub-image of G following the same steps of Algorithm 2. Then, the j -th thread of the thread-block writes the pixels values specified in its hysteresis mask ($hyst_reg$) to the j -th region of the corresponding output edges sub-image.

To write the hysteresis region, each thread accesses the output edges image through a pointer to a structure of 32 8-bit unsigned int members. As in the case of gradient regions reading, although the accesses to global memory are not coalesced, they satisfy the principle of spatial locality, and are properly aligned.

Algorithm 6: hysteresis

```
function hysteresis(// output
                  uchar d_edges[P],
                  // input
                  float d_G[P],
                  float2 d_thresholds[NS]
                  )
// Declarations
float low, high, grad_reg[32]
uint high_reg, inst_reg, hyst_reg
uchar *d_edges_reg
uchar edges_reg[32]

// Step 1: read input data
{low, high} ← read thresholds pair assigned to current
              thread-block from d_thresholds
grad_reg ← read gradient region assigned to current
            thread from d_G

// Step 2: calculate high and instability regions
high_reg ← 0
inst_reg ← 0
for i ← 0 to 31 do
  if (grad_reg[i] ≥ high) then
    set i-th bit of high_reg to 1
  else if (low < grad_reg[i] < high) then
    set i-th bit of inst_reg to 1
  end if
end for

// Step 3: calculate hysteresis region
hyst_reg ← calc_hyst_map(high_reg, inst_reg)

// Step 4: calculate edges region
for i ← 0 to 31 do
  edges_reg[i] ← i-th bit of hyst_reg
end for

// Step 5: write edges region to the output edges image
d_edges_reg ← pointer to d_edges subvector corresponding to edges region assigned to current thread
*d_edges_reg ← edges_reg
end function
```

4 Experimental evaluation

To evaluate the performance of GUD-Canny edge detection, we used the ground truth images of Heath's dataset [62], that can be downloaded from ftp://figment.csee.usf.edu/pub/Edge_Comparison/images/results/. The 28 gray reference images of this dataset were selected by humans from a limited set of edge maps, which were obtained using the Canny edge detector with different values for its parameters.

We utilized the same two candidate thresholds sets selected in [11], which were those obtained by sampling the interval [0.01, 0.25] with steps 0.01 and 0.03, and that will be denoted by $C_{0.01}$ and $C_{0.03}$, respectively.

Our test machine had a 3.50Ghz Intel Core i7-7800X CPU and 32 GB of RAM. The GPU that we used was a GeForce RTX 2080 (Turing architecture with compute capability 7.5), and no optimization flags were utilized in our implementation.

4.1 Quality evaluation

In the first experiment, we compared the quality obtained by applying Medina-Carnicer's method to the entire $W \times H$ image (classical *frame-level approach*) with the quality resulting from executing the same method on each $32 \times H$ sub-image (*distributed approach*, which is the focusing of GUD-Canny). Table 1 shows the mean-square errors (*MSE*) obtained for sets $C_{0.01}$ and $C_{0.03}$. In each row, for each candidate thresholds set, the minimum MSE is highlighted in bold. As it can be seen, the good performance of Medina-Carnicer's method not only remains in the distributed approach, but it even slightly outperforms that of frame-level approach. For the set $C_{0.01}$, the average MSEs for classical and distributed approaches were 0.0534 and 0.0498, respectively. In the case of the set $C_{0.03}$, the values were 0.0534 and 0.0502, respectively.

On the other hand, it can be observed that there is no big difference between the quality obtained using $C_{0.01}$ with respect to that resulting from utilizing $C_{0.03}$, as the average MSEs are 0.0498 and 0.0502, respectively.

4.2 Temporal efficiency evaluation

Table 2 presents the GUD-Canny edge detection times for sets $C_{0.01}$ and $C_{0.03}$. At the end of each column, statistics (average, minimum and maximum) are presented for all images, and for those of size 512×512 . Additionally, Table 3 shows the statistics of GUD-Canny speedup for $C_{0.03}$ with respect to $C_{0.01}$. From the presented results, we can see the following points:

Table 1 MSE values for frame-level Canny edge detection and distributed Canny edge detection using Medina-Carnicer's method for unsupervised determination of hysteresis thresholds

Image	Frame, $C_{0.01}$	Dist., $C_{0.01}$	Frame, $C_{0.03}$	Dist., $C_{0.03}$
Airplane (659×409)	0.0095	0.0081	0.0103	0.0071
Banana (512×468)	0.0289	0.0422	0.0310	0.0351
Basket (512×512)	0.0670	0.0603	0.0499	0.0524
Beehive (512×512)	0.0270	0.0284	0.0270	0.0278
Briefcase (577×419)	0.0237	0.0253	0.0269	0.0263
Brush (572×512)	0.0407	0.0243	0.0407	0.0259
Coffemaker (461×665)	0.0275	0.0277	0.0291	0.0289
Egg (512×512)	0.0522	0.0540	0.0534	0.0550
Elephant (512×456)	0.0523	0.0661	0.0828	0.0727
Feather (512×512)	0.0797	0.0640	0.0643	0.0624
Flower (536×509)	0.0207	0.0260	0.0249	0.0247
Golfcart (548×509)	0.0607	0.0577	0.0914	0.0721
Grater (512×438)	0.0204	0.0210	0.0252	0.0224
Mailbox (512×512)	0.0461	0.0479	0.0531	0.0550
Orange (412×472)	0.0691	0.0679	0.0676	0.0688
Pillow (552×468)	0.0394	0.0360	0.0341	0.0357
Pinecone (512×512)	0.0687	0.0629	0.0603	0.0566
Pitcher (568×419)	0.0165	0.0169	0.0195	0.0188
Pond (512×512)	0.0719	0.0724	0.0778	0.0735
Shopping cart (512×512)	0.1188	0.0761	0.0949	0.0781
Stairs (579×441)	0.0496	0.0498	0.0635	0.0540
Stapler (529×510)	0.0335	0.0373	0.0360	0.0376
Tiger (512×512)	0.1811	0.1376	0.1554	0.1270
Tire (512×512)	0.1018	0.1102	0.1018	0.1105
Traffic Cone (437×604)	0.0768	0.0636	0.0662	0.0617
Trashcan (539×433)	0.0528	0.0521	0.0528	0.0525
Turtle (512×512)	0.0142	0.0145	0.0139	0.0165
Videocamera (577×435)	0.0441	0.0445	0.0420	0.0464
Average	0.0534	0.0498	0.0534	0.0502
Minimum	0.0095	0.0081	0.0103	0.0071
Maximum	0.1811	0.1376	0.1554	0.1270

For each image and candidate thresholds set, the minimum MSE is highlighted in bold

1. GUD-Canny fully satisfies real time requirements, as its execution times are on average 1.2736 ms and 0.3637 ms for sets $C_{0.01}$ and $C_{0.03}$, respectively.
2. For the set $C_{0.03}$, the edge detection times are between 0.2814 and 0.3932 milliseconds for 512×512 images. Hence, GUD-Canny outperforms the temporal efficiency of existing GPU and FPGA implementations, like the solution of Sangeetha et al. [23], whose edge detection time is 0.672 ms for 512×512 images.
3. The speedup obtained using $C_{0.03}$ instead of $C_{0.01}$ is significant, as its values are between 2.99x and 3.90x. The reason is that the number of linking maps that have to be calculated for $C_{0.03}$ ($36 \times NS$) is much less than that for $C_{0.01}$ ($300 \times NS$). This contrasts with the small difference between the quality of edge maps obtained with these candidate thresholds sets.

4.3 Distribution of execution times

Tables 4 and 5 show the statistics (average, minimum and maximum) of kernels execution time proportions (expressed as percentages) for sets $C_{0.01}$ and $C_{0.03}$.

Unlike the case of existing GPU-based Canny edge detectors, the hysteresis stage is executed efficiently, as its average time proportions are 2.02% and 7.70% for sets $C_{0.01}$ and $C_{0.03}$, respectively.

As expected, due to their higher computational complexity, the most time-consuming operations are the calculation of matrices SM_H (whose average time proportions are 82.38% and 39.39% for sets $C_{0.01}$ and $C_{0.03}$, respectively) followed by the computation of distributions $\{P(F(x))\}$ and histograms of matrices $Prob(SM_H)$ (whose average time

Table 2 Edge detection times (ms) for candidate thresholds sets $C_{0.01}$ and $C_{0.03}$

Image	$C_{0.01}$	$C_{0.03}$
Airplane (659×409)	1.2805	0.3385
Banana (512×468)	1.0715	0.3108
Basket (512×512)	1.2234	0.3612
Beehive (512×512)	1.0518	0.3153
Briefcase (577×419)	1.2838	0.3496
Brush (572×512)	1.4066	0.3853
Coffemaker (461×665)	2.1042	0.5396
Egg (512×512)	1.0554	0.3325
Elephant (512×456)	1.0943	0.3458
Feather (512×512)	1.3306	0.3607
Flower (536×509)	1.4382	0.3703
Golfcart (548×509)	1.5631	0.4238
Grater (512×438)	1.1136	0.3106
Mailbox (512×512)	1.3599	0.3932
Orange (412×472)	0.9376	0.2671
Pillow (552×468)	1.4201	0.3870
Pinecone (512×512)	1.1283	0.3770
Pitcher (568×419)	1.2057	0.3286
Pond (512×512)	1.2288	0.3480
Shopping cart (512×512)	1.2945	0.3721
Stairs (579×441)	1.4954	0.4707
Stapler (529×510)	1.1602	0.3283
Tiger (512×512)	1.3492	0.3739
Tire (512×512)	1.2050	0.3686
Traffic Cone (437×604)	1.5956	0.4115
Trashcan (539×433)	1.0523	0.3500
Turtle (512×512)	1.0314	0.2814
Videocamera (577×435)	1.1798	0.3833
Average	1.2736	0.3637
Minimum	0.9376	0.2671
Maximum	2.1042	0.5396
Average (512×512)	1.2053	0.3531
Minimum (512×512)	1.0314	0.2814
Maximum (512×512)	1.3599	0.3932

Table 3 Statistics of GUD-Canny speedup for $C_{0.03}$ with respect to $C_{0.01}$

Images	Average	Minimum	Maximum
All	3.50x	2.99x	3.90x
512×512	3.42x	2.99x	3.69x

proportions are 7.21% and 24.87% for sets $C_{0.01}$ and $C_{0.03}$, respectively).

Figure 1 presents the statistics (average, minimum and maximum) of the total GPU time proportions corresponding to memory transferences. As it can be seen, the penalty is moderate because the percentages are less than 12% and 30% for sets $C_{0.01}$ and $C_{0.03}$, respectively.

4.4 Horizontal partitioning vs. vertical partitioning

Table 6 shows the edge detection times (ms) on 512×512 images using the candidate thresholds $C_{0.03}$ for sub-images sizes $32 \times H$ (horizontal partition) and $W \times 32$ (vertical partition). For each image, the minimum execution time is highlighted in bold. In all cases, the number of sub-images is 16 and the number of regions per sub-image is 512.

From the presented results, we can see that the execution times are slightly lower using the horizontal partitioning. The average speedup is 1.14x. As explained in Sect. 3.4.1, although the spatial locality of accesses to global memory is improved using vertical partitioning, the number of operations in the linking process is reduced if the sub-image size is $32 \times H$. Experimental evaluation has shown that the performance improvement due to the second factor is greater than that of the first.

5 Conclusions

This work has presented GUD-Canny, a novel GPU-based unsupervised and distributed implementation of Canny edge detector. Our solution overcomes the two main limitations of current Canny algorithm implementations, which are the bottleneck caused by the hysteresis process, and the use of fixed hysteresis thresholds.

Given a $W \times H$ image, GUD-Canny computes the normalized gradient magnitude, partitions it into $32 \times H$ sub-images, and calculates the optimal pair of hysteresis thresholds for each sub-image using Medina-Carnicer's method [11]. Once the hysteresis thresholds are obtained, instead of running one costly multipass CPU-GPU hysteresis process on the entire image, hysteresis thresholdings (one per sub-image, using its specific hysteresis thresholds) are executed entirely on GPU, independently and in parallel. Each thread-block performs the hysteresis process on one sub-image in shared memory, and represents each pixel of the hysteresis map with only one bit to optimize the use of the limited space of shared memory.

Experimental evaluation showed that GUD-Canny only requires 0.35 ms on average to detect edges on 512×512 images. Hence, it fully satisfies real time constraints, and is faster than existing GPU and FPGA implementations.

Table 4 Statistics of kernels execution time proportions for set $C_{0.01}$

Kernel	gauss_x (%)	gauss_y (%)	calc_grad (%)	non_max_supp (%)	calc_SM_H (%)	calc_Prob_SM_H (%)	calc_PF_histo (%)	search_thre	hyst (%)
Average	1.83	1.96	0.97	2.41	82.38	0.86	7.21	0.36	2.02
Minimum	1.33	1.34	0.69	1.67	79.81	0.53	4.85	0.21	1.50
Maximum	2.80	3.17	1.20	3.21	87.13	1.16	8.81	0.59	2.86

Table 5 Statistics of kernels execution time proportions for set $C_{0.03}$

Kernel	gauss_x (%)	gauss_y (%)	calc_grad (%)	non_max_supp (%)	calc_SM_H (%)	calc_Prob_SM_H (%)	calc_PF_histo (%)	search_thre (%)	hyst (%)
Average	6.10	6.57	3.25	8.03	39.39	2.90	24.87	1.19	7.70
Minimum	4.57	4.71	2.47	6.25	35.26	2.34	19.14	0.79	4.53
Maximum	8.93	10.17	4.04	10.35	44.57	3.96	29.55	1.48	12.62

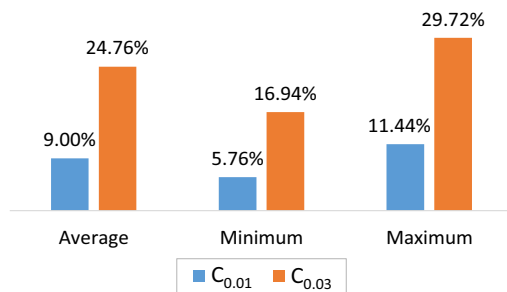


Fig. 1 Statistics of memory transferences time proportions for sets $C_{0.01}$ and $C_{0.03}$

Table 6 Edge detection times (ms) for sub-images sizes $32 \times H$ (horizontal partition) and $W \times 32$ (vertical partition)

Image	$32 \times H$	$W \times 32$
Basket (512x512)	0.3612	0.3914
Beehive (512x512)	0.3153	0.3539
Egg (512x512)	0.3325	0.4000
Feather (512x512)	0.3607	0.4054
Mailbox (512x512)	0.3932	0.4224
Pinecone (512x512)	0.3770	0.3876
Pond (512x512)	0.3480	0.4532
Shopping cart (512x512)	0.3721	0.4282
Tiger (512x512)	0.3739	0.4782
Tire (512x512)	0.3686	0.3922
Turtle (512x512)	0.2814	0.3178
Average	0.3531	0.4028
Minimum	0.2814	0.3178
Maximum	0.3932	0.4782

The candidate thresholds set is $C_{0.03}$

For each image, the minimum edge detection time is highlighted in bold

References

1. Roberts., L.: Machine perception of 3-D solids, optical and electro-optical information processing (1965)
2. Sobel, I., Feldman., G.: A 3×3 isotropic gradient operator for image processing. a talk at the Stanford Artificial Project in, 271–272 (1968)
3. Prewitt, J.M.: Object enhancement and extraction. Pict. Process. Psychopictorics **10**(1), 15–19 (1970)
4. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **6**, 679–698 (1986)
5. Gonzalez, C.I., Melin, P., Castro, J.R., Mendoza, O., Castillo, O.: An improved sobel edge detection method based on generalized type-2 fuzzy logic. Soft. Comput. **20**(2), 773–784 (2016)
6. Gunawan, T.S., Yaacob, I.Z., Kartiwi, M., Ismail, N., Za’bah, N.F., Mansor, H.: Artificial neural network based fast edge detection algorithm for mri medical images. Indones. J. Electr. Eng. Comput. Sci. **7**(1), 123–130 (2017)
7. ElAraby, W.S., Madian, A.H., Ashour, M.A., Farag, I., Nassef, M.: Fractional edge detection based on genetic algorithm. In 2017 29th International Conference on Microelectronics (ICM) (pp. 1-4). IEEE (2017, December)
8. Dagar, N.S., Dahiya, P.K.: Edge detection technique using binary particle swarm optimization. Procedia Comput. Sci. **167**, 1421–1436 (2020)
9. Sengupta, S., Mittal, N., Modi, M.: Improved skin lesion edge detection method using Ant Colony Optimization. Skin Res. Technol. **25**(6), 846–856 (2019)
10. Dhivya, R., Prakash, R.: Edge Detection Using Adaptive-Neuro-Fuzzy-Interference-System in Remote Sensing Images. J. Comput. Theor. Nanosci. **15**(9–10), 2720–2723 (2018)
11. Medina-Carnicer, R., Munoz-Salinas, R., Yeguas-Bolivar, E., Diaz-Mas, L.: A novel method to look for the hysteresis thresholds for the Canny edge detector. Pattern Recogn. **44**(6), 1201–1211 (2011)
12. Deriche, R.: Using Canny’s criteria to derive a recursively implemented optimal edge detector. Int. J. Comput. Vis. **1**(2), 167–187 (1987)
13. Torres, L., Robert, M., Bourennane, E., Paidavoine, M.: Implementation of a recursive real time edge detector using retiming techniques. In Proceedings of ASP-DAC’95/CHDL’95/VLSI’95 with EDA Technofair (pp. 811-816). IEEE (1995, August)

14. Lorca, F.G., Kessal, L., Demigny, D.: Efficient ASIC and FPGA implementations of IIR filters for real time edge detection. In Proceedings of International Conference on Image Processing (Vol. 2, pp. 406–409). IEEE (1997, October)
15. Rao, D.V., Venkatesan, M.: An efficient reconfigurable architecture and implementation of edge detection algorithm using Handle-C. In International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. (Vol. 2, pp. 843–847). IEEE (2004, April)
16. Neoh, H.S., Hazanchuk, A.: Adaptive edge detection for real-time video processing using FPGAs. *Global Signal Process.* **7**(3), 2–3 (2004)
17. Gentsos, C., Sotiropoulou, C.L., Nikolaidis, S., Vassiliadis, N.: Real-time canny edge detection parallel implementation for FPGAs. In 2010 17th IEEE International Conference on Electronics, Circuits and Systems (pp. 499–502). IEEE (2010, December)
18. He, W., Yuan, K.: An improved Canny edge detector and its realization on FPGA. In 2008 7th World Congress on Intelligent Control and Automation (pp. 6561–6564). Ieee (2008, June)
19. Li, X., Jiang, J., Fan, Q.: An improved real-time hardware architecture for Canny edge detection based on FPGA. In 2012 Third International Conference on Intelligent Control and Information Processing (pp. 445–449). IEEE (2012, July)
20. Peng, F., Lu, X., Lu, H., Shen, S.: An improved high-speed canny edge detection algorithm and its implementation on FPGA. In Fourth International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies (Vol. 8350, p. 83501V). International Society for Optics and Photonics (2012, January)
21. Abdelgawad, H.M., Safar, M., Wahba, A.M.: High level synthesis of canny edge detection algorithm on Zynq platform. *Int. J. Comput. Electr. Autom. Control Inf. Eng* **9**(1), 148–152 (2015)
22. Xu, Q., Varadarajan, S., Chakrabarti, C., Karam, L.J.: A distributed canny edge detector: algorithm and FPGA implementation. *IEEE Trans. Image Process.* **23**(7), 2944–2960 (2014)
23. Sangeetha, D., Deepa, P.: FPGA implementation of cost-effective robust Canny edge detection algorithm. *J. Real-Time Image Proc.* **16**(4), 957–970 (2019)
24. Roodt, Y., Visser, W., Clarke, W.: Image processing on the GPU: Implementing the Canny edge detection algorithm. In International Symposium of the Pattern Recognition Association of South Africa (pp. 1–6) (2007, November)
25. Luo, Y., Duraiswami, R.: Canny edge detection on NVIDIA CUDA. In 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (pp. 1–8). IEEE (2008, June)
26. Ogawa, K., Ito, Y., Nakano, K.: Efficient Canny edge detection using a GPU. In 2010 First International Conference on Networking and Computing (pp. 279–280). IEEE (2010, November)
27. Palomar, R., Palomares, J.M., Castillo, J.M., Olivares, J., Gómez-Luna, J.: Parallelizing and optimizing lip-canny using nvidia cuda. In International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (pp. 389–398). Springer, Berlin, Heidelberg (2010, June)
28. Lourenço, L. H., Weingaertner, D., Todt, E.: Efficient implementation of canny edge detection filter for ITK using CUDA. In 2012 13th Symposium on Computer Systems (pp. 33–40). IEEE (2012, October)
29. Vigil, B.M.L.P.: 2015, November. Accelerating the Canny edge detection algorithm with CUDA/GPU, International Congress COMPUMAT (2015)
30. Huang, Y., Bai, Y., Li, R., Huang, X.: Research of Canny edge detection algorithm on embedded CPU and GPU heterogeneous systems. In 2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD) (pp. 647–651). IEEE (2016, August)
31. Mogale, H.: High Performance Canny Edge Detector using Parallel Patterns for Scalability on Modern Multicore Processors. (2017) arXiv preprint [arXiv:1710.07745](https://arxiv.org/abs/1710.07745)
32. Emrani, Z., Bateni, S., Rabbani, H.: A new parallel approach for accelerating the gpu-based execution of edge detection algorithms. *J. Med. Signals Sens.* **7**(1), 33 (2017)
33. NVIDIA: CUDA Zone (2021) <https://developer.nvidia.com/category/zone-cuda-zone> <https://developer.nvidia.com/category/zone-cuda-zone>
34. Fung, J.: Computer Vision on the GPU. *GPU Gems* **2**(649–665), 34 (2005)
35. Podlozhnyuk, V.: Image convolution with CUDA. NVIDIA Corporation white paper, June, 2097(3) (2007)
36. Jourlin, M., Pinoli, J.C.: A model for logarithmic image processing. *J. Microsc.* **149**(1), 21–35 (1988)
37. Palomares, J.M., González, J., Ros, E.: Detección de bordes en imágenes con sombras mediante LIP-Canny. In Memoria del Simposio de Reconocimiento de Formas y Análisis de Imágenes del I Congreso Nacional de Informática. Granada, España. pp (pp. 71–76) (2005)
38. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: A Comparison Study of Some Configurations of the Uninorm Morphological Edge Detector. In International Conference on Fuzzy Computation Theory and Applications (Vol. 2, pp. 410–419). SCITEPRESS (2012, October)
39. González-Hidalgo, M., Massanet, S.: A fuzzy mathematical morphology based on discrete t-norms: fundamentals and applications to image processing. *Soft. Comput.* **18**(11), 2297–2311 (2014)
40. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: A new edge detector based on uninorms. In International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (pp. 184–193). Springer, Cham (2014, July)
41. Gonzalez-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: On the choice of the pair conjunction-implication into the fuzzy morphological edge detector. *IEEE Trans. Fuzzy Syst.* **23**(4), 872–884 (2014)
42. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: On the generalization of the uninorm morphological gradient. In International Work-Conference on Artificial Neural Networks (pp. 436–449). Springer, Cham (2015, June)
43. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: On the pair uninorm-implication in the morphological gradient. In Computational Intelligence (pp. 183–197). Springer, Cham (2015)
44. Bibiloni, P., González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: Mayor-torrens t-norms in the fuzzy mathematical morphology and their applications. In Fuzzy Logic and Information Fusion (pp. 201–235). Springer, Cham (2016)
45. González-Hidalgo, M., Massanet, S., Mir, A., Ruiz-Aguilera, D.: Edge image aggregation method using ordered weighted averaging functions. In 2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE) (pp. 1355–1362). IEEE (2016, July)
46. Bustince, H., Barrenechea, E., Sesma-Sara, M., Lafuente, J., Dimuro, G.P., Mesiar, R., Kolesárová, A.: Ordered directionally monotone functions: justification and application. *IEEE Trans. Fuzzy Syst.* **26**(4), 2237–2250 (2017)
47. Sun, G., Zhang, A., Ren, J., Ma, J., Wang, P., Zhang, Y., Jia, X.: Gravitation-based edge detection in hyperspectral images. *Remote Sens.* **9**(6), 592 (2017)
48. Valero, M.M., Rios, O., Mata, C., Pastor, E., Planas, E.: An integrated approach for tactical monitoring and data-driven spread forecasting of wildfires. *Fire Saf. J.* **91**, 835–844 (2017)
49. Valero, M.M., Rios, O., Pastor, E., Planas, E.: Automated location of active fire perimeters in aerial infrared imaging using unsupervised edge detectors. *Int. J. Wildland Fire* **27**(4), 241–256 (2018)

50. Sussner, P., Carazas, L.C.: An Approach Towards Image Edge Detection Based on Interval-Valued Fuzzy Mathematical Morphology and Admissible Orders. In 11th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT 2019) (pp. 690–697). Atlantis Press (2019, August)
51. Marco-Detchart, C., Bustince, H., Fernandez, J., Mesiar, R., Lafuente, J., Barrenechea, E., Pintor, J.M.: Ordered directional monotonicity in the construction of edge detectors. *Fuzzy Sets and Systems* (2020)
52. Medina-Carnicer, R., Madrid-Cuevas, F.J., Muñoz-Salinas, R., Carmona-Poyato, A.: Solving the process of hysteresis without determining the optimal thresholds. *Pattern Recogn.* **43**(4), 1224–1232 (2010)
53. Medina-Carnicer, R., Carmona-Poyato, A., Muñoz-Salinas, R., Madrid-Cuevas, F.J.: Determining hysteresis thresholds for edge detection by combining the advantages and disadvantages of thresholding methods. *IEEE Trans. Image Process.* **19**(1), 165–173 (2009)
54. Medina-Carnicer, R., Madrid-Cuevas, F.J., Carmona-Poyato, A., Muñoz-Salinas, R.: On candidates selection for hysteresis thresholds in edge detection. *Pattern Recogn.* **42**(7), 1284–1296 (2009)
55. Hancock, E.R., Kittler, J.: Adaptive estimation of hysteresis thresholds. In *Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (pp. 196–201). IEEE (1991, June)
56. Yitzhaky, Y., Peli, E.: A method for objective edge detection evaluation and detector parameter selection. *IEEE Trans. Pattern Anal. Mach. Intell.* **25**(8), 1027–1033 (2003)
57. NVIDIA: CUDA C Programming Guide (2021) <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
58. NVIDIA: CUDA C Best Practices Guide (2021) <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html> <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>
59. Luitjens, J.: “CUDA Pro Tip: Increase Performance with Vectorized Memory Access”, (Dec. 2013). <https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/>
60. Luitjens, J.: “Faster Parallel Reductions on Kepler”, (Feb. 2014). <https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/>
61. NVIDIA: CUDA Math API (2021) <https://docs.nvidia.com/cuda/cuda-math-api/index.html> <https://docs.nvidia.com/cuda/cuda-math-api/index.html>
62. Heath, M.D., Sarkar, S., Sanocki, T., Bowyer, K.W.: A robust visual method for assessing the relative performance of edge-detection algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.* **19**(12), 1338–1359 (1997)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Antonio Fuentes-Alventosa received the B.S. and M.S. degrees in Telecommunication Engineering from the University of Málaga, Spain, in 1999, the B.S. degree in Computer Science from the National Distance Education University (UNED), Spain, in 2006, and the M.S. degree in Intelligent Systems from the University of Córdoba, Spain, in 2014. Since 2001, he is software developer at Aplicaciones Informáticas Prosur in Córdoba, Spain. His research interest is in the parallelization and optimization of scientific algorithms on GPUs and heterogeneous systems.

Juan Gómez-Luna is a postdoctoral researcher at ETH Zürich. He received the BS and MS degrees in Telecommunication Engineering from the University of Sevilla, Spain, in 2001, and the PhD degree in Computer Science from the University of Córdoba, Spain, in 2012. Between 2005 and 2017, he was a lecturer at the University of Córdoba. His research interests focus on Processing-in-Memory, GPUs and heterogeneous systems, medical imaging, and bioinformatics.

R. Medina-Carnicer received the B.S. degree in Mathematics from University of Sevilla, Spain, and the Ph.D. degree in Computer Science from the Polytechnic University of Madrid, Spain, in 1992. Since 1993, he has been a lecturer of Computer Vision at Cordoba University, Spain. His research is focused on Edge Detection, 3-D Vision, Augmented Reality and Pattern Recognition.

2.3. Tercera contribución: "GVLE: a Highly Optimized GPU-Based Implementation of Variable-Length Encoding"

En este trabajo [54] se presenta GVLE, una implementación de VLE en GPU, que supera las principales limitaciones de las soluciones del estado del arte a través de las siguientes estrategias de optimización:

- El cacheo en memoria compartida de la tabla de búsqueda de palabras de código se realiza de forma que el número de conflictos de banco producidos en las búsquedas se minimiza.
- Los datos de entrada se leen mediante accesos vectorizados para aprovechar al máximo el ancho de banda de memoria global disponible.
- La codificación de cada hilo se realiza eficientemente en el espacio de registros con un alto grado de paralelismo a nivel de instrucción y un menor número de instrucciones ejecutadas.
- Se usa un nuevo método diseñado en esta tesis para la operación scan inter-bloque ejecutada en memoria global para el cálculo de las posiciones binarias de las codificaciones de bloque en el vector de salida. El mecanismo propuesto se basa en una operación scan segmentada regular ejecutada eficientemente, mediante sumas atómicas, en secuencias de longitudes binarias de 32 codificaciones de bloque consecutivas.
- Los datos de salida se escriben eficientemente en memoria global a través de accesos coalescentes.

La evaluación experimental arrojó los siguientes resultados:

- GVLE es 2.6x más rápido que la mejor implementación anterior en GPU de VLE [55].
- La operación scan es 1.62x más rápida si se usa el método scan inter-bloque propuesto en lugar del empleado en la mejor implementación anterior de VLE [55]. Por tanto, ofrece posibilidades prometedoras para acelerar algoritmos que lo requieran, como la propia operación scan y el algoritmo de compactación [56].

Esta contribución se corresponde, como resultado, con el objetivo 2.



GVLE: a highly optimized GPU-based implementation of variable-length encoding

Antonio Fuentes-Alventosa¹ · Juan Gómez-Luna² · R. Medina-Carnicer¹

Accepted: 3 December 2022

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Nowadays, the massive use of multimedia data gives to data compression a fundamental role in reducing the storage requirements and communication bandwidth. Variable-length encoding (VLE) is a relevant data compression method that reduces input data size by assigning shorter codewords to mostly used symbols, and longer codewords to rarely utilized symbols. As it is a common strategy in many compression algorithms, such as the popular Huffman coding, speeding VLE up is essential to accelerate them. For this reason, during the last decade and a half, efficient VLE implementations have been presented in the area of General Purpose Graphics Processing Units (GPGPU). The main performance issues of the state-of-the-art GPU-based implementations of VLE are the following. First, the way in which the codeword look-up table is stored in shared memory is not optimized to reduce the bank conflicts. Second, input/output data are read/written through inefficient strided global memory accesses. Third, the way in which the thread-codes are built is not optimized to reduce the number of executed instructions. Our goal in this work is to significantly speed up the state-of-the-art implementations of VLE by solving their performance issues. To this end, we propose GVLE, a highly optimized implementation of VLE on GPU, which uses the following optimization strategies. First, the caching of the codeword look-up table is done in a way that minimizes the bank conflicts. Second, input data are read by using vectorized loads to exploit fully the available global memory bandwidth. Third, each thread encoding is performed efficiently in the register space with high instruction-level parallelism and lower number of executed instructions. Fourth, a novel inter-block scan method, which outperforms those of state-of-the-art solutions, is used to calculate the bit-positions of the thread-blocks encodings in the output bit-stream. Our proposed mechanism is based on a regular segmented scan performed efficiently on sequences of bit-lengths of 32 consecutive thread-blocks encodings by using global atomic additions. Fifth, output data are written efficiently by executing coalesced global memory stores. An exhaustive experimental evaluation shows that our solution is on average 2.6× faster than the best state-of-the-art implementation. Additionally, it shows that the scan

Extended author information available on the last page of the article

algorithm is on average 1.62× faster if it utilizes our inter-block scan method instead of that of the best state-of-the-art VLE solution. Hence, our inter-block scan method offers promising possibilities to accelerate algorithms that require it, such as the scan itself or the stream compaction.

Keywords Data compression · Variable-length encoding · Huffman coding · GPU · CUDA

1 Introduction

In the current digital era, huge amounts of multimedia data, such as images and videos, are generated continuously [1]. For example, at the time of writing this paper, 720,000 hours of video are uploaded to YouTube by day [2]. Since the rate of growth of data is much higher than the rate of growth of technologies (e.g., DVDs, Blu-ray, ADSL, optical fibers, etc.), data compression has nowadays an essential role in reducing the cost of data storage and transmission [1].

Variable-length encoding (VLE) is a popular data compression method in which most frequently occurring symbols are replaced by codewords of shorter length, whereas rarely used symbols are substituted by codewords of longer length [3]. Since VLE is a common strategy in many compression algorithms [3, 4], such as the widely used Huffman coding [5, 6], acceleration of VLE is key to speed them up. In order to achieve this goal, during the last decade and a half, efficient implementations of VLE have been proposed in the area of General Purpose Graphics Processing Units (GPGPU) [7–14], which is, nowadays, mainstream high-performance computing [15–17].

The first GPGPU VLE solution is the algorithm PAVLE, proposed by Balevic [7]. This method uses an encoding alphabet of up to 256 symbols, with each symbol representing one byte. Without loss of generality, it assumes that the values and bit-lengths of the codewords are stored in a look-up table, which is cached in the on-chip shared memory. This table will be referred to as *VLET* in the rest of the paper. As GPU architectures provide more efficient support for 32-bit data types, the source and compressed data are provided and written, respectively, in two vectors of 32-bit unsigned integers. Consecutive threads load consecutive segments of elements from the source vector. Each thread uses the *VLET* for encoding the loaded segment in its private memory and calculating the corresponding bit-length. An intra-block scan primitive [18] is performed to calculate the bit-positions of the thread encodings in the corresponding thread-block encoding on the basis of their bit-lengths. The threads of a thread-block write concurrently their encodings in a buffer in shared memory using atomic operations to deal with the race conditions that occur when parts of adjacent encodings are written to the same memory location. Once the writing is finished, the content of the buffer is copied to the output vector at the same position of the corresponding source segment in the input vector. After the encoding is finished, a second kernel is launched to compact the output vector. Experimental evaluation showed speedups with respect to the serial implementation on a

2.66 GHz Intel QuadCore CPU of up to 35x. Fuentes-Alventosa et al. [8] presented CUVLE, a new implementation of VLE on CUDA. As in the case of PAVLE, the VLET is cached in shared memory, and consecutive threads process consecutive source segments. However, their approach uses the following optimization strategies. First, persistent blocks [19], which equals the grid size to the maximum number of resident thread-blocks, thereby minimizing the number of VLET loads in shared memory. Second, contiguous writing of thread-block encodings in global memory, which avoids the necessity of running any compaction extra kernel. The bit-positions of the thread encodings in the output vector are calculated by combining the efficient intra-block scan algorithm of Sengupta et al. [20] with the adjacent thread-block synchronization mechanism proposed by Yan et al. [21]. Third, direct writing of thread-block encodings in global memory. Since CUVLE does not use an intermediate buffer in shared memory, it saves the time to make additional operations, avoids the appearance of bank conflicts and saves the reserved space for the buffer. Experimental evaluation showed that CUVLE is on average more than 20 and 2 times faster than the corresponding CPU serial implementation and PAVLE, respectively. The test machine had a 2.67GHz Intel Core i7 920 CPU and 12 GB of RAM, and the GPUs utilized were a GeForce GT 640 2GB GDDR5 and a GeForce GTX 550 Ti. Rahmani et al. [9] proposed a CUDA-based Huffman coder that does not have any constraint on the maximum code bit-length by generating an intermediate byte stream where each byte represents a single bit of the compressed output stream. After the Huffman tree generation is done serially on the CPU, the encoding is performed in parallel on the GPU following the next three steps (each one implemented with a different kernel). First, the code offsets for each input symbol in the intermediate stream are calculated using the scan method presented in [18]. Second, the intermediate stream is generated by the i -th thread of the second kernel writing the code of the i -th input symbol to its corresponding memory slots in the intermediate stream. Third, the output stream is obtained by each thread of the third kernel reading 8 consecutive bytes from the intermediate stream, and generating a single byte of the output stream. As the encoding is implemented with three kernels, this solution has two main overheads: the extra long latency global memory accesses required to transmit intermediate results between kernels, and the costly launches and terminations of the kernels. Experimental evaluation on the NVIDIA GTX 480 GPU showed speedups with respect to the CPU serial implementation of up to 22x on an Intel Core 2 Quad CPU running at 2.40 GHz. The work of Yamamoto et al. [10] focused on GPU acceleration of Huffman encoding and decoding and was developed in CUDA. As in the case of CUVLE, the VLE stage is implemented with only one kernel. The authors exposed that their kernel is similar to CUVLE, but it is much more faster because, instead of using Yan et al.'s mechanism [21], it utilizes a novel adjacent thread-block synchronization method, which is much more efficient. The reason is that, in the Yan et al.'s algorithm [21], each thread-block looks back the result written in global memory by only one thread-block, while, in the Yamamoto et al.'s approach [10], each thread-block looks back 32 previous results simultaneously. Experimental evaluation for ten files on NVIDIA Tesla V100 GPU showed that Yamamoto et al.'s VLE implementation is between 2.87 and 7.70 times

faster than CUVLE. For this reason, the best state-of-the-art implementation of VLE on GPU is the solution of Yamamoto et al.

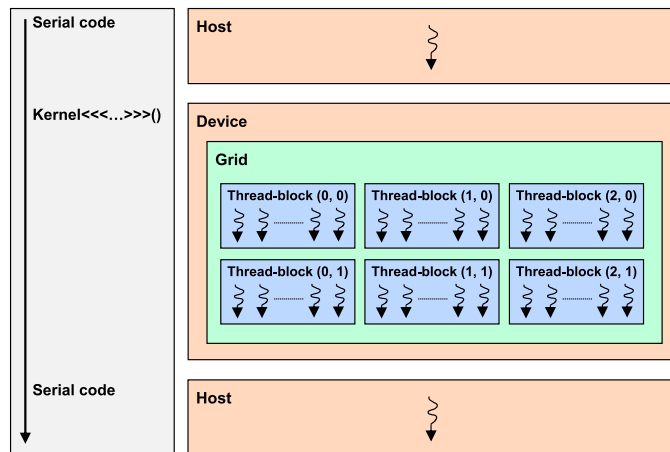
The main performance issues of the state-of-the-art GPU-based implementations of VLE [8, 10] are the following. First, the way in which the VLET is cached is not optimized to reduce the shared memory bank conflicts. Second, each thread reads/writes the elements of its input/output segment one by one, which results in inefficient strided global memory accesses. Third, the way in which the thread-codes are built is not optimized to reduce the number of executed instructions. In order to solve these issues, we propose GVLE, a highly optimized implementation of VLE on GPU, which significantly speeds up the solution of Yamamoto et al. As in previous approaches [7, 8, 10], the VLET is cached in shared memory, and consecutive threads process consecutive segments of the input vector. However, GVLE uses the following optimization strategies. First, the VLET storage in shared memory is done in a way that minimizes the bank conflicts. Second, the input segments are read by using vectorized loads to exploit fully the available global memory bandwidth [22]. Third, each thread, after reading its assigned segment, encodes it efficiently in the register space with high instruction-level parallelism and lower number of executed instructions. Fourth, a novel inter-block scan method, which outperforms those of Yan et al. [21] and Yamamoto et al. [10], is used to calculate the bit-positions of the thread-blocks encodings in the output vector. Our proposed mechanism is based on a regular segmented scan performed efficiently on sequences of bit-lengths of 32 consecutive thread-blocks encodings by using global atomic additions [23]. Fifth, the thread-block encodings are written efficiently to the output vector by executing coalesced global memory stores [24].

Our main contributions in this work are the following:

- A highly optimized GPU-based approach to VLE, called GVLE,¹ that significantly improves the state-of-the-art implementations [8, 10].
- A novel inter-block scan method for calculating the bit-positions of thread-blocks encodings that outperforms those used in [8] and [10].
- A comparison of our solution with the best state-of-the-art implementation [10]. An exhaustive experimental evaluation shows that our proposal is on average 2.6× faster than the method presented in [10].
- A comparison of our inter-block scan method with that of [10]. The experimental results show that the speedup of the scan operation using our inter-block scan algorithm is on average 1.62× with respect to using the method of [10].

The rest of the paper is organized as follows. Section 2 gives background for CUDA, VLE and the Yamamoto et al.'s implementation of VLE [10]. Section 3 presents GVLE and compares our method with the one proposed in [10], so that the achieved performance improvement can be clearly established. Section 4 shows the experimental evaluation of our algorithm and a comparison to the method of Yamamoto

¹ The source code is available at <https://github.com/z12fuala/GVLE>.

Fig. 1 Execution of a CUDA program

et al. [10], CUVLE [8] and the serial implementation of VLE. Section 5 reviews related work. Finally, the main conclusions are stated in Section 6.

2 Background

This section is structured in the following way. Section 2.1 gives a brief overview of CUDA, and cites several relevant documents that can provide further background to readers. Section 2.2 defines VLE, and highlights its important role in data compression. Section 2.3 gives a detailed description and a critical analysis of the best state-of-the-art implementation of VLE on GPU ([10]).

2.1 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing framework developed by NVIDIA for GPGPU computing [16]. Since its release in 2007, thousands of applications have been developed on CUDA [15, 16], so it is one of the main responsible technologies for the GPGPU computing revolution.

CUDA greatly facilitates to developers the implementation of parallel algorithms by providing a small set of extensions to popular languages such as C, C++, Fortran, Python and MATLAB [16]. In this work, we have utilized CUDA C++.

As shown in Fig. 1 [23], in a CUDA program, the sequential parts runs on the CPU (usually referred to as the *host*), while the compute intensive parts are executed by thousands of threads on the GPU (commonly named as the *device*). The functions executed on the GPU are called *kernels*, which are defined by the programmer using the `__global__` declaration specifier. The number of threads that execute a kernel is specified using the `<<<...>>>` *execution configuration* syntax. They are organized into one-, two- or three-dimensional blocks of threads, which are called *thread-blocks*. A kernel is executed by a set of identical thread-blocks, called *grid*, which can also have up to three dimensions.

The architecture of a CUDA GPU is composed of a set of *streaming multiprocessors* (SMs) [23]. When a kernel is launched, the thread-blocks of the grid are

distributed to the available multiprocessors with execution capacity. The threads of each thread-block run concurrently on a single multiprocessor, and each multiprocessor can concurrently execute many thread-blocks. As the execution of the thread-blocks finishes, new ones are launched in the available multiprocessors. The SMs execute the threads in groups of 32 called *warps*. The threads of a warp start at the same program address, but each one has its own instruction address counter and register state, and, hence, they are free to branch and execute independently. Although CUDA developers can ignore this behavior for the correctness of their applications, they can greatly improve their performance by minimizing the warp divergence.

CUDA threads can access three types of memory spaces during their execution [23]:

- Each thread has private memory consisting of *registers* and *local memory*. Its lifetime is that of the thread.
- Each thread-block has *shared memory* visible to all threads in it. Its lifetime is that of the thread-block. The `__shared__` qualifier is used for the declaration of variables in shared memory.
- All threads of a grid have access to a read/write *global memory*, and two other read memories: the *constant memory*, used to store non-modifiable values, and the *texture memory*, optimized for accesses with 2D spatial locality. The contents of these memories are persistent between the different kernel calls of the same application.

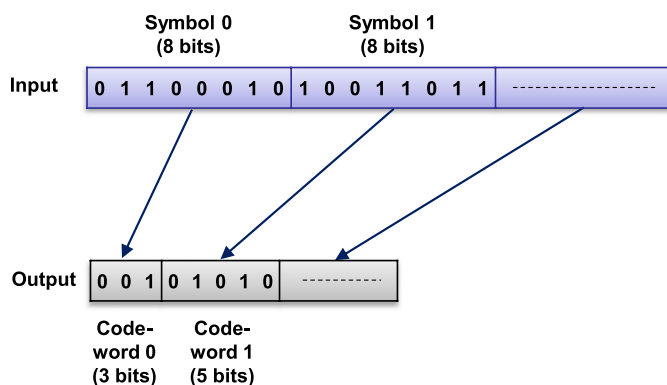
Global memory is the most abundant of these memory spaces [24]. On the other hand, global, local, and texture memory have the highest access latency, followed by constant memory, shared memory, and registers [24]. A very important optimization technique is the coalescing of global memory accesses [23, 24]. When a warp performs an operation on global memory, the memory accesses of its threads are coalesced into one or more memory transactions according to the size of the accessed words and the distribution of the memory addresses. The more scattered the accesses are, the more transactions are necessary, and, hence, the more reduced the throughput is.

2.2 Variable-length encoding (VLE)

Input data to a compression algorithm can be modeled as a sequence of elements, called *symbols*, belonging to an alphabet [4]. A symbol can be an ASCII character, a byte, an audio sample, etc. Given an alphabet $S = \{s_0, s_1, \dots, s_{n-1}\}$, its digital representation is called the *code* $C = \{c_0, c_1, \dots, c_{n-1}\}$, and the representation c_i of each symbol is called the *codeword* for symbol s_i . Codes are classified into *fixed length codes (FLC)* and *variable-length codes (VLC)*, depending on the length of their codewords is fixed or variable, respectively. The process of assigning codewords to symbols of input data is called *encoding*, and the reverse process is called *decoding*.

Variable-length encoding (VLE) [3] is a compression method in which input data size is reduced by using a VLC that assigns shorter codewords to mostly used

Fig. 2 Variable-Length Encoding (VLE). Input data size is reduced by assigning shorter codewords to mostly used symbols, and longer codewords to rarely utilized symbols



symbols, and longer codewords to rarely utilized symbols. Figure 2 illustrates VLE. For example, consider the alphabet $S = \{A, B, C, D, E\}$ and the 9-symbols input data string $BAAAAAAC$ [4]. On the one hand, if the encoding is performed using a 3-bit FLC, the bit-length of the result is $9 \times 3 = 27$. On the other hand, if the encoding is performed using the VLC $C = \{0, 100, 101, 110, 111\}$, the bit-length of the result is $1 \times 3 + 7 \times 1 + 1 \times 3 = 13$, which is less than half that obtained with the FLC.

VLE is one of the main building blocks in many compression algorithms [3, 4], such as the popular Huffman coding [5, 6], which is the most relevant entropy coding method at present [25]. Huffman coding is a component of the Deflate algorithm, which is used in the file compression programs ZIP, 7ZIP, GZIP, and PKZIP, and in the image compression format PNG, for example [26]. Additionally, Huffman coding is the most used entropy coding algorithm in multimedia encoding standards such as JPEG, MPEG, H.264 and VC-1 [27], and is a critical step in an increasing number of high-performance computing applications [12, 28, 29]. Since VLE is an essential step in so many important present and future compression algorithms, its acceleration is fundamental to speed them up.

2.3 Solution of Yamamoto et al

The best state-of-the-art implementation of VLE on GPU is the solution presented by Yamamoto et al. [10], which was developed in CUDA. It is composed of only one kernel, which will be referred to as *YAVLE* in the rest of the paper. In this section, we give a detailed description of *YAVLE* based on the paper of Yamamoto et al. [10], and the source code of their solution published on GitHub at github.com/daisuke-takafuji/Huffman_coding_Gap_arrays.

YAVLE operates on 8-bit symbols and, therefore, it utilizes an alphabet of up to 256 symbols. The VLET is provided in a vector (d_VLET) of 256 elements, whose base type is a structure (*Codeword*) with two 32-bit unsigned int members that represent the value and the bit-length of a codeword. Yamamoto et al. assume that the maximum bit-length of codewords is 16 because Huffman coding with this limited maximum codeword can be generated efficiently [10, 30, 31]. In fact, actually, the maximum codeword length is limited in the most implementations of Huffman coding [10].

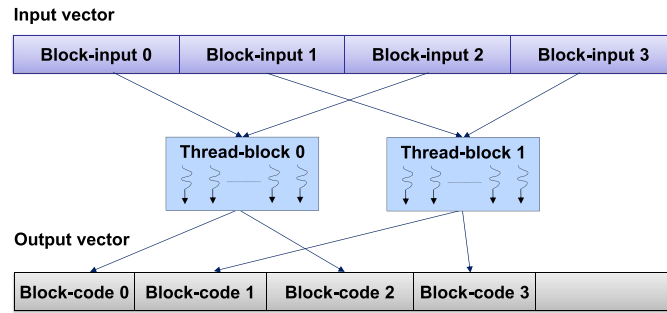


Fig. 3 Inter-block mechanism for an input vector of 4 block-inputs and a grid of 2 thread-blocks. The thread-block 0 processes the block-inputs 0 and 2, and writes the corresponding block-codes 0 and 2 in the output vector. The thread-block 1 performs the same actions with the block-inputs 1 and 3

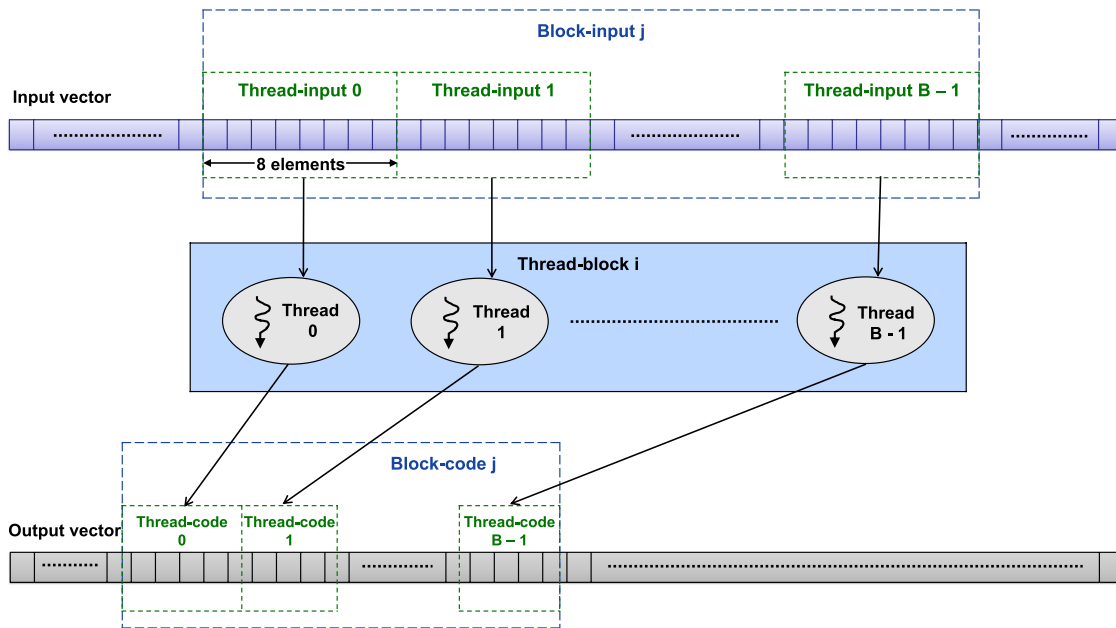


Fig. 4 Intra-block mechanism. The thread-block i (of size B) processes the block-input j and writes the corresponding block-code j in the output vector. Each thread k of the thread-block i encodes the thread-input k and writes the thread-code k in the output vector

The source data are supplied in a vector of 32-bit unsigned integers (d_{input}). Hence, each element contains 4 symbols. The compressed data are written in an output vector (d_{output}) of 32-bit unsigned integers too. Let B be the thread-block size. The vector d_{input} is partitioned into segments of size $B \times 8$, which will be named as *block-inputs*. Each block-input is processed by a thread-block and the encoding result, which will be referred to as *block-code*, is written in d_{output} . Figure 3 illustrates this inter-block mechanism. Consecutive threads of each thread-block process consecutive segments of eight elements (i.e., 32 symbols) of the corresponding block-input, which will be named as *thread-inputs*. The encoding of a thread-input will be referred to as *thread-code*. Figure 4 clarifies this intra-block mechanism.

Algorithm 1 provides a high-level description of YAVLE. Each thread-block caches the VLET in shared memory (Step 1) and encodes a different subset of block-inputs (Steps 2 to 7). The number of thread-blocks of the grid is set to

the maximum number of resident thread-blocks. Hence, the number of VLET loads in shared memory is minimized. The indexes of the block-inputs are obtained from a zero-initialized global counter (Steps 2 and 7). The function *get_global_counter_value* uses the CUDA function *atomicAdd* [23] to return the successive values of the global counter, that is, 0, 1, 2, and so on. This mechanism ensures that, when a thread-block starts the processing of a block-input i , the management of block-inputs 0, 1, ..., $i - 1$ have already begun. The processing of each block-input consists of Steps 3 to 6.

Algorithm 1: YAVLE algorithm

```

1 kernel YAVLE(// output
2     uint *d_output,
3     // input
4     uint *d_input,
5     Codeword d_VLET[256],
6     ull *d_scan
7 )
8     // Step 1: cache VLET in shared memory
9     Codeword s_VLET[256] ← cache_VLET(d_VLET)
10
11    // Step 2: get index of first block-input to encode
12    uint blockinput_idx ← get_global_counter_value()
13
14    // While there are block-inputs to encode...
15    while (blockinput_idx < num_blockinputs)
16        // Step 3: calculate index of thread-input
17        uint thinput_idx ← blockinput_idx × blockDim.x +
18                            threadIdx.x
19
20        // Step 4: calculate bit-length of thread-code
21        uint thcode_len ← YAVLE_calc_thcode_len(d_input,
22                                                thinput_idx,
23                                                s_VLET)
24
25        // Step 5: calculate bit-position of thread-code
26        // in output vector
27        ull thcode_pos ← YAVLE_calc_thcode_pos(thcode_len,
28                                                d_scan)
29
30        // Step 6: write thread-code to output vector
31        YAVLE_write_thcode(d_output, thcode_pos,
32                            d_input, thinput_idx, s_VLET)
33
34        // Step 7: get index of next block-input to encode
35        blockinput_idx ← get_global_counter_value()
36    end while
37 end kernel

```

In Sects. 2.3.1 to 2.3.4, we describe Steps 4 to 6 of Algorithm 1, respectively.

2.3.1 Calculation of bit-lengths of thread-codes

In Step 4 of Algorithm 1, each thread iterates over the 32 symbols of its thread-input to calculate the bit-length $thcode_len$ (line 12) of the corresponding thread-code. For each symbol, if the element to which it belongs has not been loaded from d_input yet, that element is read. Then, the symbol is extracted from its element, and its codeword is obtained from the VLET. The bit-length of the thread-code is computed by accumulating the bit-lengths of its codewords.

2.3.2 Calculation of bit-positions of thread-codes in output vector

In Step 5 of Algorithm 1, the bit-position $thcode_pos$ (line 15) of each thread-code in the output vector is computed by adding the bit-position of the thread-code in its block-code ($pos_of_thcode_in_bcode$) to the bit-position of the block-code in the output vector ($blockcode_pos$).

On the one hand, the intra-block scan method of Sengupta et al. [20] is performed on the parameters $thcode_len$ of the current block-code to compute the corresponding bit-positions $pos_of_thcode_in_bcode$ and the bit-length of the block-code ($blockcode_len$).

On the other hand, a novel inter-block scan algorithm [10], which is described in the next section, is executed on the parameters $blockcode_len$ of the block-codes to compute their bit-positions $blockcode_pos$.

2.3.2.1 Yamamoto et al.'s inter-block scan method The inter-block scan method of Yamamoto et al. uses an auxiliary global vector d_scan (line 5) of 64-bit unsigned integers, whose initial values are zero. The number of elements of d_scan equals to the number of block-codes, and each element i is assigned to the block-code i . Each value written in d_scan has two mutually exclusive flags, named as A and P , which are located in bits 56 and 63, respectively. Given an element $d_scan[i]$, if the flag A is set, then it stores the bit-length of block-code i ; otherwise, if the flag P is set, it holds the sum of bit-lengths of block-codes 0 to i , which is the bit-position of the block-code $i + 1$ in the output vector.

Given a block-code i , the first warp of its assigned thread-block computes the parameter $blockcode_pos$ by following the steps presented in Algorithm 2. If the block-code is the first, $blockcode_pos$ is clearly zero (line 3). Otherwise, the warp iterates on the necessary 32-elements segments of d_scan before to $d_scan[i]$ to compute $blockcode_pos$ (lines 8 to 20). This parameter is obtained by the sum of the elements $d_scan[k]$, $d_scan[k + 1]$, ... $d_scan[i - 1]$, where k is the index of the last element previous to $d_scan[i]$ with the flag P activated.

Algorithm 2: Yamamoto et al.'s inter-block scan method to compute the parameter *blockcode_pos* of each block-code *i*

```

1  if i = 0 then // Block-code i is the first
    // Write the bit-position of block-code 1,
    // which equals to the bit-length of block-code 0,
    // in d_scan
2  d_scan[i] ← SET_FLAG_P(blockcode_len)
    // Since the block-code is the first, its bit-position
    // is clearly 0
3  blockcode_pos ← 0
4  else // Block-code i is not the first
    // Write the bit-length of block-code i
    // in d_scan
5  d_scan[i] ← SET_FLAG_A(blockcode_len)
    // Initialize the bit-position of block-code i
6  blockcode_pos ← 0
    // Compute the indexes of the elements of the first
    // 32-elements segment before d_scan[i] (i.e., the
    // segment composed of d_scan[i-32], d_scan[i-31],
    // ... d_scan[i-1])
7  int j ← i - 1 - warp_lane
8  while true
    // Read repeatedly the current 32-elements
    // segment until all their elements are non-zero
9  ull aux ← d_scan[j]
10 while aux = 0 and j > -1
11     aux ← d_scan[j]
12 end while
    // Perform a warp reduction to compute the sum of
    // the elements k, k + 1, ... 31 of the current
    // segment, where k is 0, if no element has the
    // flag P activated, or the index of the last
    // element with the flag P activated, otherwise
13 ull prev_32_blockcodes_sum ← warp_reduction(aux)
    // Update the bit-position of block-code i
14 blockcode_pos ← blockcode_pos +
15     DEL_FLAG(prev_32_blockcodes_sum)
    // Exit the loop if the bit-position of the
    // block-code i has already been calculated
16 if FLAG_P_IS_SET(prev_32_blockcodes_sum) then
17     exit while
18 end if
    // Compute the indexes of the elements of the next
    // 32-elements segment before d_scan[i]
19 j ← j - 32
20 end while
    // Write the bit-position of the next block-code in
    // d_scan
21 ull next_blockcode_pos ← blockcode_pos + blockcode_len
22 d_scan[i] ← SET_FLAG_P(next_blockcode_pos)
23 end if

```

The main difference between CUVLE [8] and YAVLE is in the method used to calculate the parameters *blockcode_pos*. Yamamoto et al.'s inter-block scan is much more efficient than Yan et al.'s technique [21] employed by CUVLE because, to obtain the bit-position of a block-code i , the first processes rapidly 32-element segments previous to $d_scan[i]$, whose values are read simultaneously by the first warp of the thread-block, while the second iterates only over one previous element ($d_scan[i - 1]$). This optimization is the unique reason of the significant speedup of YAVLE with respect to CUVLE [10].

2.3.3 Writing of thread-codes to output vector

In Step 6 of Algorithm 1, each thread iterates over the 32 symbols of its thread-input in the same way as it does to calculate the bit-length of the thread-code (Step 4). Let d_thcode be a pointer to the first element of d_output that will be occupied by the thread-code. As the codewords assigned to the symbols are obtained from the VLET, their bits are concatenated in a 32-bit variable ($word_val$) and their bit-lengths added in a second 32-bit variable ($word_len$) while the bit-length of the resulting encoding is less than or equal to 32. When the last condition is not satisfied, the first 32 bits of the resulting encoding are written in the corresponding element of d_thcode , and the value and bit-length of the remaining encoding are stored in $word_val$ and $word_len$, respectively. The process continues until all the codewords are written. To avoid race conditions with the previous and next thread-codes, the first and last writes are performed by using atomic OR operations [23].

3 Highly optimized GPU-based implementation of VLE (GVLE)

In this section, we present *GVLE*, our GPU-based implementation of VLE, which has been developed using the popular NVIDIA CUDA framework [16]. It is also compared with Yamamoto et al.'s proposal so that the achieved performance improvement can be clearly established.

As previous solutions [8, 10], *GVLE* is composed of only one kernel, whose execution configuration sets the number of thread-blocks to the maximum number of resident thread-blocks. The inputs and outputs of *GVLE* are the same as those of YAVLE, except that, in the case of *GVLE*, the VLET is provided in two separate vectors, one of 256 16-bit unsigned integers (d_VLET_val) and the other of 256 8-bit unsigned integers (d_VLET_len), which store the values and the bit-lengths of the codewords, respectively. As in the case of YAVLE, it is assumed that the maximum bit-length of codewords is 16.

Algorithm 3 presents the pseudo code of *GVLE*. As in previous approaches [8, 10], each thread-block caches the VLET in shared memory (Step 1) and encodes a different subset of block-inputs (Steps 2 to 9). The technique used to get the indexes of the block-inputs (Steps 2 and 9) is the same as that of YAVLE (Sect. 2.3). Let us define a *warp-code* as the encoding of the 32 thread-inputs processed by a warp, that is to say, the concatenation of the thread-codes computed by a warp. The processing of each block-input consists of Steps 3 to 8.

In Sect. 3.1, we describe Step 1 of Algorithm 3, and, in Sects. 3.2 to 3.6, Steps 4 to 8, respectively.

Algorithm 3: GVLE algorithm

```

1  kernel GVLE(// output
2      uint *d_output,
3      // input
4      uint *d_input,
5      ushort d_VLET_val[256],
6      uchar d_VLET_len[256],
7      ull *d_scan
8  )
9      // Step 1: cache VLET in shared memory
10     ushort s_VLET_val[256] ← d_VLET_val
11     uchar s_VLET_len[256] ← d_VLET_len
12
13     // Step 2: get index of first block-input to encode
14     uint blockinput_idx ← get_global_counter_value()
15
16     // While there are block-inputs to encode...
17     while (blockinput_idx < num_blockinputs)
18         // Step 3: calculate index of thread-input
19         uint thinput_idx ← blockinput_idx × blockDim.x +
20                             threadIdx.x
21
22         // Step 4: read thread-input
23         uchar32 thinput ← ((uchar32 *)d_input)[thinput_idx]
24
25         // Step 5: calculate thread-code
26         uint seg_val[16], seg_len[16], thcode_len
27         GVLE_calc_thcode(// output
28             seg_val, seg_len, thcode_len,
29             // input
30             thinput, s_VLET_val, s_VLET_len)
31
32         // Step 6: calculate parameters of warp-code
33         uint pos_of_thcode_in_wcode, wcode_len, wcode_pos
34         GVLE_calc_wcode_param(
35             // output
36             pos_of_thcode_in_wcode, wcode_len, wcode_pos,
37             // input
38             thcode_len, d_scan)
39
40         // Step 7: build warp-code in shared memory
41         uint s_warpcode[num_warps][513]
42         GVLE_build_warpcode(// output
43             s_warpcode[warp_idx],
44             // input
45             seg_val, seg_len, thcode_len,
46             pos_of_thcode_in_wcode, wcode_pos)
47
48         // Step 8: write warp-code to output vector
49         GVLE_write_warpcode(// output
50             d_output,
51             // input
52             wcode_pos, s_warpcode[warp_idx],
53             wcode_len)
54
55         // Step 9: get index of next block-input to encode
56         blockinput_idx ← get_global_counter_value()
57     end while
58 end kernel

```

3.1 VLET caching

In Step 1 of Algorithm 3, since the VLET is used intensively for searching the codewords, each thread-block caches it in the fast on-chip shared memory. The global memory vectors d_VLET_val (line 8) and d_VLET_len (line 9) are copied to the identical shared memory vectors s_VLET_val and s_VLET_len , respectively, in a fully coalesced way.

The warp accesses to the VLET are random because they depend on the source data. For this reason, in order to minimize the bank conflicts caused by irregular warp accesses [23, 24], the VLET is implemented with two separate vectors (lines 8 and 9) whose base types have the minimum size necessary to store codewords of up to 16 bits (16-bits for s_VLET_val and 8-bits for s_VLET_len). In contrast, as YAVLE caches the VLET in a single vector whose base type (*Codeword*) has a size of 64-bits, the number of bank conflicts is much higher. The reason is that, in the case of YAVLE, each codeword is stored in two consecutive 32-bits elements of shared memory, while, in the case of GVLE, two codewords' values are cached in one 32-bit element, and four codewords' bit-lengths are kept in one 32-bit element.

On the other hand, although GVLE has to access two vectors (instead of one, as YAVLE) to get the value and the bit-length of one codeword, these readings are fast because they are executed in parallel at the instruction level.

3.2 Reading of thread-inputs

In Step 4 of Algorithm 3, each thread reads the 32 symbols of its thread-input through one vectorized access using the custom vector type *uchar32*, which is composed of 32 8-bit unsigned integers, and stores the thread-input in the variable *thinput* (line 14). Vectorized loads are an important CUDA optimization because they increase bandwidth and reduce both instruction count and latency [22].

In contrast, YAVLE reads the elements of its thread-input one by one, which results in inefficient strided global memory accesses [23, 24]. In addition, YAVLE, instead of loading its thread-input from global memory once, reads it twice: the first time to calculate the bit-length of the thread-code, and the second time to compute the bit-stream of the thread-code on the fly during its writing in the output vector.

3.3 Calculation of thread-codes

In Step 5 of Algorithm 3, each thread searches in s_VLET_val and s_VLET_len the codewords assigned to the 32 symbols stored in *thinput* to compute the corresponding thread-code (lines 16 to 18). Since a thread-code is the concatenation of 32 consecutive codewords and the bit-length of each codeword is no more than 16-bits, a thread-code is made up of 16 binary segments (each segment i corresponding to the concatenation of the codewords $2 \times i$ and $2 \times i + 1$), whose bit-lengths are not greater than 32-bits. Taking this into account, the values and the bit-lengths of the segments are calculated and cached in the private arrays *seg_val* and *seg_len* (line

15), of 16 32-bit unsigned integers each, respectively. Additionally, the bit-length $thcode_len$ of each thread-code (line 15) is obtained by adding the bit-lengths of its segments.

The calculation of thread-codes is efficient for the following reasons. First, there are no dependencies between the different computations of segments, hence the degree of instruction-level parallelism is high. Second, each segment calculation is performed with few operations of high throughput (one binary shift and two sums). Third, there is no warp divergence in the computation of segments. Fourth, the arrays seg_val and seg_len are placed in the register space [24] because (1) they are small, (2) they are indexed with constant quantities, and (3) the kernel does not use more registers than available.

Since YAVLE computes the bit-stream of each thread-code by concatenating its 32 codewords on the fly during its writing to the output vector, the number of executed instructions is higher than that of GVLE.

3.4 Calculation of parameters of warp-codes

In Step 6 of Algorithm 3, each thread-block calculates the following parameters, which are necessary for the posterior processing of the warp-codes of the current block-code (line 19):

- Bit-position of each thread-code in its warp-code ($pos_of_thcode_in_wcode$).
- Bit-length of each warp-code ($wcode_len$).
- Bit-position of each warp-code in the output vector ($wcode_pos$).

On the one hand, the intra-block scan method of Sengupta et al. [20] is executed on the bit-lengths of the thread-codes of the current block-code to calculate the parameters $pos_of_thcode_in_wcode$, $wcode_len$, the bit-position of each warp-code in the block-code ($pos_of_wcode_in_bcode$), and the bit-length of the block-code ($blockcode_len$).

On the other hand, the bit-position of each block-code in the output vector ($blockcode_pos$) is obtained by carrying out a scan operation on the bit-lengths of the block-codes using a novel inter-block scan algorithm, which is proposed in the next section.

Once a warp gets the parameters $blockcode_pos$ and $pos_of_wcode_in_bcode$, it computes $wcode_pos$ by adding them.

3.4.1 Our inter-block scan method

In our algorithm, the global vector d_scan (line 6), whose elements are initially zero, is used to perform a regular segmented inclusive scan on segments of bit-lengths of 32 consecutive block-codes. Each segment i is composed of the bit-lengths of block-codes $32 \times i$ to $32 \times i + 31$, and its prefix sum is written in the corresponding 32-elements sub-vector i of d_scan . The scan of each segment is performed by a set of 32 thread-blocks, which will be referred to as *sub-grid*.

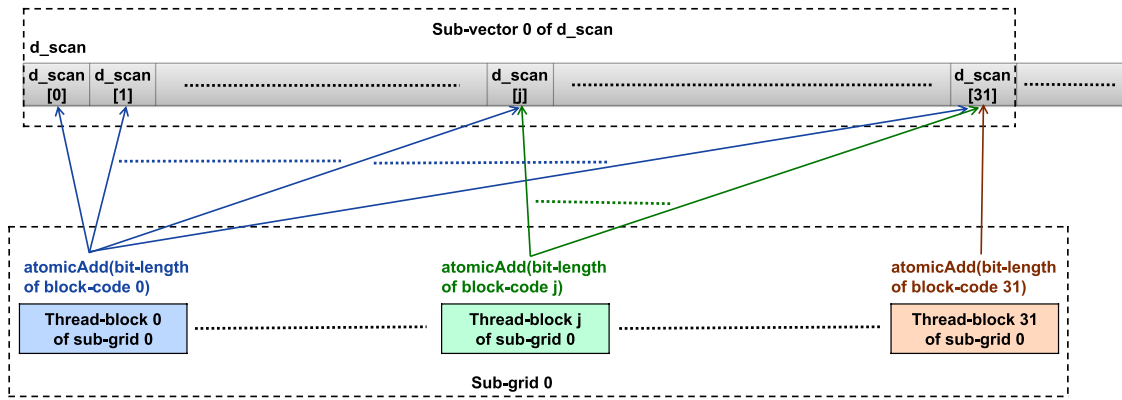


Fig. 5 GVLE inter-block scan on the first segment

Each segment i is processed by a sub-grid i , which is composed of the thread-blocks that manage the block-codes $32 \times i$ to $32 \times i + 31$. The scan of each segment i is calculated directly in the d_scan sub-vector i by the first warp of each thread-block j of the sub-grid i performing an atomic addition [23] of the bit-length of its block-code to the elements j to 31 of the sub-vector i . Note that the number of atomic additions carried out by each sub-grid on each element j of the corresponding sub-vector is $j + 1$. Figure 5 illustrates this mechanism for the first segment.

In order to detect that the segmented prefix sum has already been calculated for a particular d_scan sub-vector element, bits 57 to 62 of each element are used to store the number of atomic additions performed on it. This sums counter is implemented by performing the atomic additions with the bit 57 of the bit-lengths of the block-codes set to 1. The bits 0 to 56 of each element j of each d_scan sub-vector i are used to store the corresponding segmented scan value, i.e., the sum of bit-lengths of block-codes $32 \times i$ to $32 \times i + j$. In the case of the thirty-second element of each sub-vector i except the first, a second value is assigned to its bits 0 to 56 in a posterior stage of our algorithm, which is the not-segmented scan value, i.e., the sum of bit-lengths of block-codes 0 to $32 \times i + 31$. To distinguish between these two mutually exclusive values, in the second case, a flag will be activated in the bit 63, which will be referred to as *flag P*. Table 1 shows an example of GVLE inter-block scan, which presents an extract of the first 64 values written in d_scan (i.e., the corresponding to the first two segments). Note that $d_scan[63]$ is the only element that has the flag P activated (the bit 63 is 1). The reason is that it stores the sum (1, 206, 728) of all the bit-lengths of segments 0 and 1. The remaining elements hold the segmented scan value (bits 0 to 56), and the number of atomic additions performed on them (bits 57 to 62).

Let us define a *sub-code* as the bit-stream composed of the block-codes managed by a sub-grid. Given a thread-block j of a sub-grid i , the first warp of the thread-block follows the next steps to calculate the parameter *blockcode_pos* of the corresponding block-code:

1. It performs an atomic addition of the bit-length of the block-code (with the bit 57 set to 1) to the elements j to 31 of the sub-vector i .

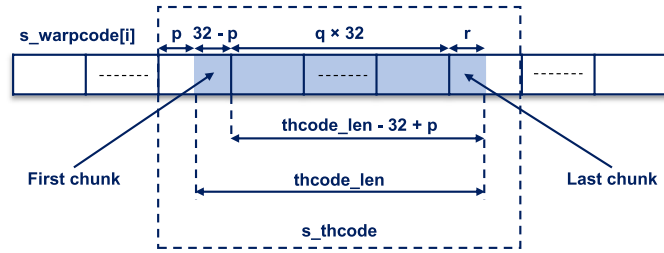
Table 1 Example of GVLE inter-block scan on the two first segments

i	Bit-length of block-code i	Value stored in bits 57 to 63 of $d_scan[i]$	Value stored in bits 0 to 56 of $d_scan[i]$
0	19,036	1	19,036
1	18,641	2	37,677
2	18,210	3	55,887
.	.	.	.
29	18,976	30	565,395
30	19,209	31	584,604
31	19,026	32	603,630
32	18,775	1	18,775
33	19,223	2	37,998
34	19,331	3	57,329
.	.	.	.
61	18,632	30	565,170
62	18,865	31	584,035
63	19,063	64	1,206,728

2. It gets the bit-position of the block-code in its sub-code, which will be referred to as $pos_of_bc_in_sc$, in the following way. If $j = 0$, clearly $pos_of_bc_in_sc$ is 0. Otherwise, it reads repeatedly the element $j - 1$ of the sub-vector i until its sums counter is j . The parameter $pos_of_bc_in_sc$ is obtained by resetting the bits 57 to 62.
3. It gets the bit-position of the sub-code in the output vector, which will be referred to as $pos_of_sc_in_out$, in the following way. If $i = 0$, clearly $pos_of_sc_in_out$ is 0. Otherwise, it reads repeatedly the element 31 of the sub-vector $i - 1$ until its sums counter is 32 or the flag P is activated (i.e., the value stored in bits 57 to 63 is 32 or 64). If the flag P of the read value is not activated, it repeats the same procedure on sub-vectors $i - 2, i - 3, \dots$ until the read value has the flag P activated or there are no more sub-vectors to process. The parameter $pos_of_sc_in_out$ is obtained by accumulating the read values, with the bits 57 to 63 set to 0, as they are read.
4. The parameter $blockcode_pos$ is obtained by adding $pos_of_bc_in_sc$ to $pos_of_sc_in_out$.
5. If $j = 31$ (i.e., the thread-block is the last of the sub-grid):
 - (a) It computes the bit-length of the sub-code i (sc_len) by adding $pos_of_bc_in_sc$ to $blockcode_len$.
 - (b) It computes the sum of $pos_of_sc_in_out$ to sc_len , and stores it in the element 31 of the sub-vector i with the flag P activated. Note that the written value is the bit-position of the sub-code $i + 1$ in the output vector.

As will be shown in Sect. 4, our inter-block scan method outperforms that of Yamamoto et al. The reasons are the following:

Fig. 6 Writing of a thread-code in the shared memory buffer $s_warp_code[i]$



1. Since the scan on different segments are executed independently by the corresponding sub-grids, the degree of parallelism is higher in our algorithm.
2. In YAVLE, each thread-block assigns the bit-length of its block-code to a single element of d_scan , while, in GVLE, each thread-block uses the bit-length of its block-code to update $32 - j$ elements of d_scan , where j is the index of the thread-block within its sub-grid. As the number of elements updated by thread-blocks $0, 1, \dots, 31$ of a sub-grid are $32, 31, \dots, 1$, respectively, the average number of elements updated per thread-block is 16.
3. In GVLE, the scan of each segment is performed directly on its sub-vector by using atomic operations. In contrast, in YAVLE, after writing the bit-lengths on d_scan , it is necessary to read them in groups of 32 elements to perform the scan operation.

3.5 Building of warp-codes in shared memory

In Step 7 of Algorithm 3, each warp i of each thread-block builds its warp-code in the shared memory buffer $s_warp_code[i]$ (line 23) of 513 32-bit unsigned integers. The warp-code is written right-shifted the same number of bits that it will be in the target sub-vector of d_output . The size of each buffer is 513 for the following reasons. On the one hand, since the bit-length of each codeword is no more than 16-bits, the number of codewords of each thread-code is 32, and the warp size is 32, the maximum number of bits of a warp-code is $16 \times 32 \times 32 = 16,384$ bits, which can be stored in $16,384/32 = 512$ unsigned integers. On the other hand, as each warp-code is written right-shifted, an extra unsigned integer is necessary, so the size of each warp buffer is $512 + 1 = 513$.

The warp-code is built in the buffer by each thread of the warp writing its thread-code, which was cached previously in the private arrays seg_val and seg_len ((lines 15 to 18)), in the bit-position of the thread-code within its warp-code. Given a thread-code, let s_thcode be the buffer sub-vector in which it is written, p the bit-position of the thread-code in s_thcode , and $thcode_len$ the bit-length of the thread-code. We call q and r the quotient and the remainder of the division of $(thcode_len - 32 + p)$ by 32, respectively. As shown in Fig. 6, the first $32 - p$ bits of the thread-code (which will be referred to as *first chunk*) are written right-aligned in $s_thcode[0]$, the following q 32-bits sequences in $s_thcode[1], \dots, s_thcode[q]$, and, if $r > 0$, the last r bits (which will be denoted by *last chunk*) in $s_thcode[q + 1]$. The writing of the warp-code is carried out by following the next steps:

1. Each thread writes all the bits of its thread-code, except the last chunk, in the elements $s_thcode[0], \dots s_thcode[q]$.
2. All threads of the warp synchronize by executing the CUDA function `__syncwarp` [23].
3. Each thread, if its thread-code has a last chunk, writes it in the first r bits of $s_thcode[q + 1]$.

Note that the warp synchronization ensures that no race conditions exist in the writing of those elements of the buffer in which the last chunk of a thread-code and the first chunk of the next thread-code are stored.

3.6 Writing of warp-codes to output vector

In Step 8 of Algorithm 3, each warp, after writing its warp-code in the shared memory buffer, iterates over 32-elements segments of the buffer to copy them to the target d_output sub-vector in a coalesced way [24] (lines 28 to 31). The first and last elements of the target d_output sub-vector are written using atomic OR operations ([23]) to preserve the last chunk of the previous warp-code and the first chunk of the next warp-code, respectively, if they have already been written.

In YAVLE, each thread writes the elements of its thread-code directly to global memory one by one, which results in inefficient strided global memory accesses.

4 Experimental evaluation

To evaluate GVLE and compare it to YAVLE, we have used the Standard Canterbury Corpus, consisting of 11 files (alice29.txt, asyoulik.txt, cp.html, fields.c, grammar.lsp, kennedy.xls, lcet10.txt, plravn12.txt, ptt5, sum, xargs.1) and the Large Canterbury Corpus, consisting of 3 files (bible.txt, e.coli, world192.txt), which are available at <http://www.data-compression.info/Corpora/CanterburyCorpus/>. Furthermore, to fully utilize the resources of the target GPU, we have increased each of the 14 files by replicating its original content the minimum number of times necessary to make the final size greater than or equal to 100 megabytes.

The method used to compute the VLETs is the Huffman coding, and we have obtained the implementation of YAVLE from the source code of Yamamoto et al.'s solution, published on GitHub at github.com/daisuke-takafuji/Huffman_coding_Gap_arrays.

In order to measure precisely the execution times of the kernels, we run one warm-up iteration and then fifty iterations to report their statistical values.

Our test machine has a 3.50Ghz Intel Core i7-7800X CPU, 32 GB of RAM, and a GeForce RTX 2080 GPU (Turing architecture with compute capability 7.5). The CUDA toolkit and the GPU driver versions are 11.1 and 512.15, respectively. We have used the default optimization flag (`-O3`) [32].

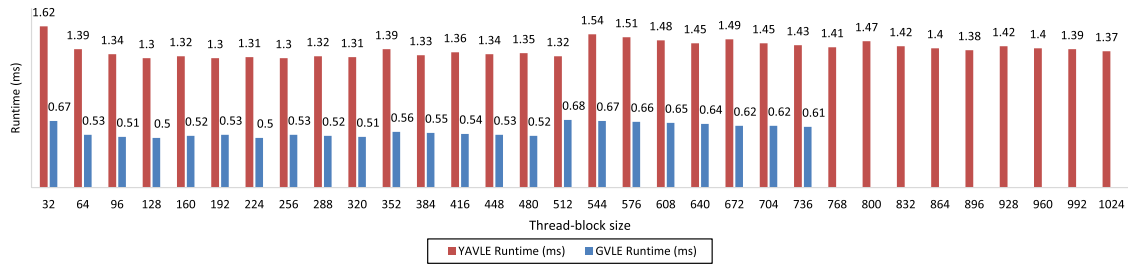


Fig. 7 YAVLE and GVLE runtimes for thread-block sizes between 32 and 1024 (in steps of 32 threads)

Table 2 Number of shared memory load bank conflicts and runtimes of kernels YAVLE and EXP_VLET, and corresponding improvements in EXP_VLET over YAVLE

Parameter	YAVLE	EXP_VLET	Improvement
Shared load bank conflicts	13,839,817	3,262,067	4.24×
Runtime (ms)	1.30	1.07	1.21×

4.1 Sensitivity analysis of the thread-block size

To analyze the effect of the thread-block size on the performance of YAVLE and GVLE, we have measured their average runtimes for all possible values of the thread-block size that are multiples of 32. Figure 7 shows the obtained results. Note that, in the case of GVLE, the maximum thread-block size is 736, due to the shared memory buffer used to build the warp-codes.

As it can be seen, the effect of the thread-block size on the performance of YAVLE and GVLE is low. Since 128 is an optimal thread-block size for both GVLE and YAVLE, we have used this value in the remaining experiments.

4.2 Comparison of GVLE with YAVLE

In order to determine the contribution of each of our optimizations in the performance improvement in GVLE with respect to YAVLE, we have developed a set of kernels that starting from YAVLE, gradually implement the different optimization techniques of GVLE. In the following sections, we present the obtained results.

4.2.1 VLET implementation

We have built the kernel *EXP_VLET* from YAVLE by substituting YAVLE's implementation of VLET (i.e., one vector of 256 elements of type *Codeword*) for that of GVLE (i.e., a vector of 256 16-bit unsigned integers to store the values of the codewords, and a second vector of 256 8-bit unsigned integers to hold the bit-lengths of the codewords).

Table 3 Number of global load/reduction/store transactions, number of executed instructions and runtimes of kernels EXP_VLET and EXP_GM, and corresponding improvements in EXP_GM over EXP_VLET

Parameter	EXP_VLET	EXP_GM	Improvement
Global load transactions	60,039,145	7,164,487	8.38×
Global reduction transactions	4,307,360	205,459	20.96×
Global store transactions	7,156,553	2,231,864	3.21×
Executed instructions	1,153,782.69	882,368.02	1.31×
Runtime (ms)	1.07	0.70	1.53×

Table 4 Number of executed instructions and runtimes of kernels EXP_GM and EXP_REG, and corresponding improvements in EXP_REG over EXP_GM

Parameter	EXP_GM	EXP_REG	Improvement
Executed instructions	882,368.02	542,291.61	1.63×
Runtime (ms)	0.70	0.57	1.23×

As shown in Table 2, EXP_VLET is 1.21× faster than YAVLE due to the improvement in the shared memory load bank conflicts (4.24×).

4.2.2 Global memory reading and writing

We have obtained the kernel *EXP_GM* from EXP_VLET by replacing its global memory reading and writing methods for those of GVLE. On the one hand, instead of reading the thread-inputs element by element through strided global memory accesses, they are read through vectorized accesses using the custom vector type *uchar32*. On the other hand, instead of writing the thread-codes directly to global memory element by element through strided global memory accesses, each warp, after writing its warp-code in its shared memory buffer, iterates over 32-elements segments of the buffer to copy them to global memory in a coalesced way.

As shown in Table 3, EXP_GM is 1.53× faster than EXP_VLET due to the improvement in the global load transactions (8.38×), the global reduction transactions (20.96×), the global store transactions (3.21×) and the executed instructions (1.31×).

4.2.3 Thread-codes building

We have developed the kernel *EXP_REG* from EXP_GM by performing the following two changes. First, the calculation of the bit-length of the thread-code (Step 4 of Algorithm 1) is replaced by the complete calculation of the thread-code (Step 5 of Algorithm 3). Second, each warp-code is built in shared memory by concatenating the 16-binary segments of each thread-code (Step 7 of Algorithm 3), instead of by linking the 32 codewords of each thread-code (Step 6 of Algorithm 1).

Table 5 Number of global atomic/load/store transactions, number of executed instructions and runtimes of kernels EXP_REG and GVLE, and corresponding improvements in GVLE over EXP_REG

Parameter	EXP_REG	GVLE	Improvement
Global atomic transactions	26,051	141,610	0.18×
Global load transactions	7,259,723	6,721,148	1.08×
Global store transactions	2,231,864	2,181,301	1.02×
Executed instructions	542,291.61	517,852.75	1.05×
Runtime (ms)	0.57	0.50	1.14×

Table 6 Number of shared memory load bank conflicts, number of global load/reduction/store transactions, number of executed instructions and runtimes of kernels YAVLE and GVLE, and corresponding improvements in GVLE over YAVLE

Parameter	YAVLE	GVLE	Improvement
Shared load bank conflicts	13,839,817	3,170,669	4.36×
Global load transactions	60,261,119	6,721,148	8.97×
Global reduction transactions	4,307,361	205,459	20.96×
Global store transactions	7,156,553	2,181,301	3.28×
Executed instructions	1,319,851.51	517,852.75	2.55×
Runtime (ms)	1.30	0.50	2.57×

As shown in Table 4, EXP_REG is 1.23× faster than EXP_GM due to the improvement in the executed instructions (1.63×).

4.2.4 Inter-block scan method

The unique difference between the kernels EXP_REG and GVLE is that the former uses the Yamamoto et al.'s inter-block scan method (Sect. 2.3.3), while the latter uses our inter-block scan algorithm (Sect. 3.4.1).

As shown in Table 5, although the number of global atomic transactions of EXP_REG is 0.18× that of GVLE, GVLE is 1.14× faster than EXP_REG due to the improvement in the global load transactions (1.08×), the global store transactions (1.02×) and the executed instructions (1.05×).

4.2.5 Global contribution of our optimization strategies

Table 6 compares YAVLE and GVLE by presenting the values of the performance parameters referenced in previous sections. As it can be seen, GVLE is 2.57× faster than YAVLE due to the improvement in the shared memory load bank conflicts (4.36×), the global load transactions (8.97×), the global reduction transactions (20.96×), the global store transactions (3.28×), and the executed instructions (2.55×). Finally, Fig. 8 presents the runtimes of YAVLE and GVLE, and Table 7 the corresponding statistics. As it can be seen, the acceleration of GVLE is significant, since its value is between 1.97× and 3.11×.

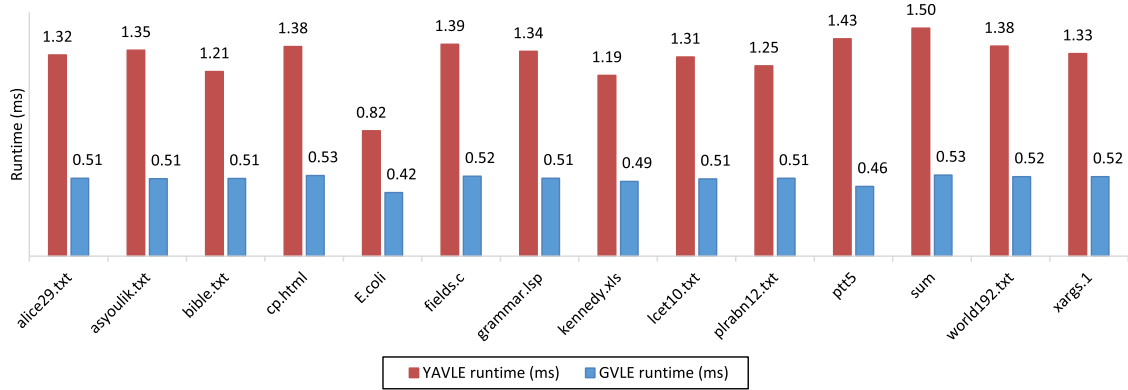


Fig. 8 YAVLE and GVLE runtimes for each test file

Table 7 Statistics of GVLE, YAVLE, CUVLE and CPU_VLE Runtimes, and of GVLE Speedups

	Average	Minimum	Maximum
GVLE runtime (ms)	0.50	0.42	0.53
YAVLE runtime (ms)	1.30	0.82	1.50
Speedup of GVLE	2.57×	1.97×	3.11×
CUVLE runtime (ms)	6.85	6.59	7.36
Speedup of GVLE	13.63×	12.93×	16.67×
CPU_VLE runtime (ms)	191.75	112.07	225.93
Speedup of GVLE	377.15×	273.15×	425.90×

4.3 Comparison between GVLE, CUVLE and the serial implementation of VLE

Table 7 compares the statistics of GVLE, CUVLE and the implementation of VLE on CPU (*CPU_VLE*). As it can be seen, GVLE is on average 13.63× faster than CUVLE, our previous implementation of VLE, which represents a significant advance in our research on GPU-based acceleration of VLE. On the other hand, the speedup of VLE with respect to the serial implementation of VLE is considerable, since it is on average 377.15×.

4.4 Comparison between inter-block scan methods

In order to compare the performance of our inter-block scan method with those of Yamamoto et al. [10] and Yan et al. [21], we have developed three kernels that perform the scan operation using the method of Sengupta et al. [20] for the intra-block scan, and one of the methods under study for the inter-block scan. We call the kernels that use our inter-block scan method, that of Yamamoto et al, and that of Yan et al., *GVLE_scan*, *YAVLE_scan* and *CUVLE_scan*, respectively. The input and output vectors are the same as those used in our previous experiments, with the

Table 8 Statistics of kernels GVLE_scan, YAVLE_scan and CUVLE_scan Runtimes, and of GVLE_scan Speedups

	Average	Minimum	Maximum
GVLE_scan runtime (ms)	1.22	1.20	1.27
YAVLE_scan runtime (ms)	1.97	1.95	2.05
Speedup of GVLE_scan	1.62×	1.56×	1.65×
CUVLE_scan runtime (ms)	47.13	46.68	48.02
Speedup of GVLE_scan	38.32×	36.87×	39.49×

particularity that each thread, instead of reading eight consecutive elements of the input vector, reads only one.

Table 8 compares the statistics of the kernels. As it can be seen, GVLE_scan clearly outperforms YAVLE_scan, since the speedup is between 1.56× and 1.65×. Moreover, the speedup of GVLE_scan_with respect to CUVLE_scan is very high, as it is between 36.87× and 39.49×. It can be seen that although our inter-block scan algorithm is the least influential optimization in the acceleration of YAVLE, it provides a significant speedup in the case of the scan operation. Therefore, it can be used to accelerate significantly algorithms that require performing an inter-block scan, such as the scan operation itself or the stream compaction [33].

5 Related work

In this section, we review some GPU-based solutions in which VLE is partially implemented, since it only operates on small data chunks, and does not concatenate the resulting encodings. Each data chunk is mapped to a thread-block [11, 12], a warp [13] or even a thread [13, 14]. In the corresponding compression algorithms, the concatenation is not necessary [11, 12] or is implemented with a separate component [13].

Tian et al. [11] proposed cuSZ, a CUDA-based implementation of SZ [34], which is one of the best error-bounded lossy compressors for scientific data. cuSZ splits the whole dataset into multiple chunks, and compresses them independently, which favors coarse grained decompression. A dual-quantization scheme is applied to completely remove the strong data dependency in SZ's prediction-quantization step. The quantization codes generated by the dual-quantization procedure are compressed by a customized Huffman coding, which follows the next four steps. First, calculate the statistical frequency for each quantization bin (as a symbol) using the method proposed by Gómez-Luna et al. [35]. Second, build the Huffman tree based on the frequencies and construct a base codebook. Third, transform the base codebook to the canonical Huffman codebook [36]. Fourth, encode in parallel according to the codebook, and concatenate Huffman codes into a bitstream (called deflating). Experimental evaluation on a NVIDIA V100 GPU showed that cuSZ improves SZ's compression throughput by up to 13.1x over the production version running on two 20-core Intel Xeon Gold 6148 CPUs.

In a later work, Tian et al. [12] presented an efficient CUDA-based Huffman encoder that outperforms the scheme presented in [11]. The main novelties of this work are the following. First, the development of an efficient parallel codebook construction on GPUs that scales effectively with the number of input symbols. Second, a novel reduction-based encoding scheme that can efficiently merge the codes on GPU. Experimental evaluation showed that their solution can improve the encoding throughput by up to 5.0x and 6.8x on NVIDIA RTX 5000 and V100 GPUs, respectively, over their previous proposal [11], and by up to 3.3 over the multithread encoder on two 28-core Xeon Platinum 8280 CPUs.

Zhu et al. [13] presented an efficient parallel entropy coding method (EPent), which was implemented in CUDA, to accelerate the entropy coding stage of JPEG image compression algorithm. EPent has three phases: coding, shifting and stuffing. In the coding phase, the 8×8 blocks of quantized transformed coefficients are encoded in parallel, via run-length encoding and Huffman coding, to form their corresponding bitstreams. In the shifting phase, the bitstreams are shifted to ensure that the bitstreams of adjacent coefficient blocks can be correctly concatenated. Finally, in the stuffing phase, the output stream is produced by concatenating the shifted bitstreams. Experimental evaluation on a NVIDIA GTX 1050Ti GPU showed that compared with sequential implementation on a 2.4 GHz i7-4700HQ CPU, the maximum speedup ratio of entropy coding can reach 39 times without affecting compressed images quality.

Fuentes-Alventosa et al. [14] proposed CAVLCU, an efficient implementation of CAVLC on CUDA, which was based on four key ideas. First, CAVLCU is composed of only one kernel to avoid the long latency global memory accesses required to transmit intermediate results between different kernels, and the costly launches and terminations of additional kernels. Second, the efficient Yan et al.'s synchronization mechanism [21] is used for thread-blocks that process adjacent frame regions (in horizontal and vertical dimensions) to share results in global memory space. Third, the available global memory bandwidth is exploited fully by using vectorized loads to move directly the quantized transform coefficients to registers. Fourth, register tiling is used to implement the zigzag sorting, thus obtaining high instruction-level parallelism. Experimental evaluation on NVIDIA GPUs GeForce GTX 970 and GeForce RTX 2080 showed that CAVLCU is between 2.5x and 5.4x faster than the best previous GPU-based implementation of CAVLC [37, 38].

6 Conclusions

This work has presented GVLE, a highly optimized GPU-Based implementation of variable-length encoding. Our solution overcomes the main performance issues of the state-of-the-art GPU-based implementations of VLE by using the next optimization strategies. First, the caching of the codeword look-up table is done in a way that minimizes the bank conflicts. Second, input data is read by using vectorized loads to exploit fully the available global memory bandwidth. Third, each thread encoding is performed efficiently in the register space with high instruction-level parallelism and lower number of executed instructions.

Fourth, a novel inter-block scan method, which outperforms those of state-of-the-art solutions, is used to calculate the bit-positions of the thread-blocks encodings in the output bit-stream. Our proposed mechanism is based on a regular segmented scan performed efficiently on sequences of bit-lengths of 32 consecutive thread-blocks encodings by using global atomic additions. Fifth, output data are written efficiently by executing coalesced global memory stores.

An exhaustive experimental evaluation shows that our solution is between $1.97\times$ and $3.11\times$ faster than the best state-of-the-art implementation due to the improvement in the shared memory load bank conflicts ($4.36\times$), the global load transactions ($8.97\times$), the global reduction transactions ($20.96\times$), the global store transactions ($3.28\times$) and the executed instructions ($2.55\times$). Moreover, experimental results show that the speedup of the scan operation using our inter-block scan algorithm is on average $1.62\times$ and $38.32\times$ with respect to using the methods of Yamamoto et al. and Yan et al., respectively. Therefore, our method can be used to accelerate significantly algorithms that require performing an inter-block scan, such as the scan operation itself or the stream compaction.

Acknowledgements Not applicable.

Author Contributions All authors contributed to the study conception. Design, data collection and analysis were performed by AF-A. The first draft of the manuscript was written by AF-A and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding None.

Data availability Not applicable.

Declarations

Conflict of interest No, I declare that the authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Ethical approval and consent to participate The corresponding author has read the Springer journal policies on author responsibilities and submits this manuscript in accordance with those policies.

Consent for publication I have read and understood the publishing policy, and submit this manuscript in accordance with this policy.

References

1. Jayasankar U, Thirumal V, Ponnuram D (2021) A survey on data compression techniques: from the perspective of data quality, coding schemes, data type and applications. *J King Saud Univ-Comput Inf Sci* 33(2):119–140
2. Wise J “How many videos are uploaded to youtube a day in 2022?”, June 2022. <https://earthweb.com/how-many-videos-are-uploaded-to-youtube-a-day/>
3. Banerji A, Ghosh AM (2010) *Multimedia technologies*. Tata McGraw Hill, New Delhi
4. Pu IM (2005) *Fundamental data compression*. Butterworth-Heinemann
5. Huffman DA (1952) A method for the construction of minimum-redundancy codes. *Proc IRE* 40(9):1098–1101

6. Moffat A (2019) Huffman coding. *ACM Comput Surv (CSUR)* 52(4):1–35
7. Balevic A (2009) Parallel variable-length encoding on GPGPUs. In *European Conference on Parallel Processing*, Springer, Berlin, Heidelberg. pp 26–35
8. Fuentes-Alventosa A, Gómez-Luna J, González-Linares JM, Guil N (2014) CUVLE: variable-length encoding on CUDA. In *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on IEEE*. pp 1–6
9. Rahmani H, Topal C, Akinlar C (2014) A parallel Huffman coder on the CUDA architecture. In *2014 IEEE Visual Communications and Image Processing Conference, IEEE*. pp 311–314
10. Yamamoto N, Nakano K, Ito Y, Takafuji D, Kasagi A, Tabaru T (2020) Huffman coding with gap arrays for GPU acceleration. In *49th International Conference on Parallel Processing-ICPP*. pp 1–11
11. Tian J, Di S, Zhao K, Rivera C, Fulp MH, Underwood R, Cappello F (2020) Cusz: an efficient gpu-based error-bounded lossy compression framework for scientific data. *arXiv preprint arXiv:2007.09625*
12. Tian J, Rivera C, Di S, Chen J, Liang X, Tao D, Cappello F (2021) Revisiting huffman coding: toward extreme performance on modern gpu architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE*. pp 881–891
13. Zhu F, Yan H (2022) An efficient parallel entropy coding method for JPEG compression based on GPU. *J Supercomput* 78(2):2681–2708
14. Fuentes-Alventosa A, Gómez-Luna J, González-Linares JM, Guil N, Medina-Carnicer R (2022) CAVLCU: an efficient GPU-based implementation of CAVLC. *J Supercomput* 78(6):7556–7590
15. NVIDIA: GPU-Accelerated Applications (2020) <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/gpu-applications-catalog.pdf>
16. NVIDIA: CUDA Zone (2022) <https://developer.nvidia.com/category/zone/cuda-zone>
17. Khronos group: OpenCL (2022) <https://www.khronos.org/opencl/>
18. Harris M, Sengupta S, Owens JD (2007) Parallel prefix sum (scan) with CUDA. *GPU Gems* 3(39):851–876
19. Martín PJ, Ayuso LF, Torres R, Gavilanes A (2012) Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In *2012 International Conference on High Performance Computing & Simulation (HPCS), IEEE*. pp 511–519
20. Sengupta S, Harris M, Garland M (2008) Efficient parallel scan algorithms for GPUs. NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003, 1(1), 1–17
21. Yan S, Long G, Zhang Y (2013) StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* pp 229–238
22. Luitjens J “CUDA Pro Tip: increase Performance with Vectorized Memory Access”, Dec. 2013. <https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/>
23. NVIDIA: CUDA C Programming Guide (2022) <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
24. NVIDIA: CUDA C Best Practices Guide (2022) <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>
25. Manz O (2021) *Well Packed-Not a Bit Too Much*. Springer Fachmedien Wiesbaden
26. Gyasi-Agyei A (2019) *Telecommunications engineering: principles and practice*. World Scientific, Singapore
27. Unger H, Kyamaky K, Kacprzyk J. (Eds.). (2011). *Autonomous Systems: Developments and Trends (Vol. 391)*. Springer
28. Lal S, Lucas J, Juurlink B (2017) E²MC: entropy encoding based memory compression for GPUs. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE*. pp 1119–1128
29. Choukse E, Sullivan MB, O’Connor M, Erez M, Pool J, Nellans D, Keckler SW (2020) Buddy compression: enabling larger memory for deep learning and HPC workloads on GPUs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), IEEE*. pp 926–939
30. Larmore LL, Hirschberg DS (1990) A fast algorithm for optimal length-limited Huffman codes. *J ACM (JACM)* 37(3):464–473

31. Katajainen J, Moffat A, Turpin A (1995) A fast and space-economical algorithm for length-limited coding. In International Symposium on Algorithms and Computation. Springer, Berlin, Heidelberg. pp 12–21
32. NVIDIA CUDA Compiler Driver NVCC (2022) <https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html>
33. Luna JGG, Chang LW, Sung IJ, Hwu WM, Guil N (2015) In-place data sliding algorithms for many-core architectures. In: 2015 44th International Conference on Parallel Processing, IEEE. pp 210–219
34. Di S, Cappello F (2016) Fast error-bounded lossy HPC data compression with SZ. In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE. pp 730–739
35. Gómez-Luna J, González-Linares JM, Benavides JI, Guil N (2013) An optimized approach to histogram computation on GPU. *Mach Vis Appl* 24(5):899–908
36. Barnett ML (2003) U.S. Patent No. 6,657,569. Washington, DC: U.S. Patent and Trademark Office
37. Su H, Zhang C, Chai J, Wen M, Wu N, Ren J (2011) A high-efficient software parallel CAVCL encoder based on GPU, 2011 34th International Conference on Telecommunications and Signal Processing (TSP), Budapest, pp 534–540
38. Su H, Wen M, Wu N, Ren J, Zhang C (2014) Efficient parallel video processing techniques on GPU: from framework to implementation. *Sci World J*, 2014

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Antonio Fuentes-Alventosa¹ · Juan Gómez-Luna² · R. Medina-Carnicer¹

✉ Antonio Fuentes-Alventosa
antonio.fa@gmail.com

Juan Gómez-Luna
juang@ethz.ch

R. Medina-Carnicer
rmedina@uco.es

¹ Department of Computer Sciences and Numerical Analysis, University of Córdoba, Córdoba, Spain

² Department of Information Technology and Electrical Engineering, ETH Zürich, Zürich, Switzerland

3. Conclusiones y futuros trabajos

3.1. Conclusiones

La realización de esta tesis ha conducido a la optimización en GPU de un conjunto de algoritmos científicos (CAVLC, detector de bordes de Canny y VLE) y a la obtención de un nuevo método scan inter-bloque en GPU eficiente. Las conclusiones son las siguientes:

1. En [23] se ha presentado CAVLCU, una implementación optimizada de CAVLC entre 2.5x y 5.4x más rápida que la implementación que representaba el estado del arte [13].
2. En [36] se ha propuesto GUD-Canny, un detector de bordes de Canny no supervisado y distribuido, que es más rápido que las implementaciones en GPU [18, 37, 38, 39, 40, 41, 42, 43] y en FPGA [45, 46, 47, 48, 49, 50, 51, 52, 53] existentes, y cumple los siguientes requisitos de diseño:
 - Resolución de las principales limitaciones de las implementaciones del algoritmo de Canny, que son el cuello de botella causado por el proceso de histéresis y el uso de umbrales de histéresis fijos.
 - Determinación no supervisada de los umbrales de histéresis mediante el método propuesto en [44].
 - Ejecución en tiempo real, al ser 0.35 ms el tiempo promedio en detectar los bordes de imágenes 512x512.
3. En [54] se ha presentado GVLE, una implementación optimizada de VLE 2.6x más rápida en promedio que la mejor solución anterior [55]. Además, se ha propuesto un nuevo método eficiente para la operación scan inter-bloque que se ejecuta en memoria global para calcular las posiciones binarias de las codificaciones de bloque en el vector de salida. En el caso de la operación scan, si se usa este método en lugar del empleado en la mejor implementación anterior de VLE [55], se logra una aceleración de 1.62x.

3.2. Futuros trabajos

Los resultados obtenidos en esta tesis sugieren nuevas ideas a explorar, tales como:

1. **Uso de CAVLCU en la optimización de codificadores de vídeo e imagen.** Teniendo en cuenta el uso masivo de la compresión de datos multimedia en la presente era digital [57, 58], CAVLCU se puede aprovechar en futuros trabajos sobre implementaciones en GPU de los siguientes sistemas:
 - Codificadores H.264 software que requieran funcionalidad inexistente en los codificadores H.264 hardware incorporados en las tarjetas gráficas [24], como encriptación de datos [25, 26, 27, 28] y ocultación de información [29, 30, 31, 32].
 - Codificadores de vídeo e imágenes en otros formatos distintos a H.264, como imágenes médicas [33, 34, 35].
2. **Utilización de GUD-Canny en la optimización de aplicaciones de vídeo e imagen.** Dada la esencial importancia de la detección de bordes en diferentes ámbitos (como el procesamiento de imágenes, la visión por computador y el reconocimiento de patrones), GUD-Canny se puede aplicar en la optimización en GPU de sistemas de tiempo real que requieran detección de bordes no supervisada.
3. **Uso de GVLE en la optimización de sistemas de compresión de datos.** Al ser VLE uno de los principales bloques de construcción de muchos sistemas de compresión, como la popular codificación de Huffman, GVLE se puede aprovechar en futuros trabajos sobre implementaciones en GPU de dichos sistemas.
4. **Utilización del método scan inter-bloque de GVLE en la optimización de algoritmos que requieran la operación scan.** Dada la aceleración de 1.62x lograda en la operación scan usando el método scan inter-bloque de GVLE en lugar del empleado en la mejor implementación anterior de VLE, resulta de interés estudiar:
 - La posibilidad de obtener una versión optimizada de la operación scan a través del método scan inter-bloque propuesto y de un conjunto de técnicas de optimización, a determinar, del componente intra-bloque de scan.
 - La aplicación del método scan inter-bloque propuesto en la aceleración de algoritmos que requieren la operación scan, como el algoritmo de compactación [56].

Referencias

- [1] Dehal, R. S., Munjal, C., Ansari, A. A., & Kushwaha, A. S. (2018, October). GPU Computing Revolution: CUDA. In 2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN) (pp. 197-201). IEEE.
- [2] Kirk, D. B., & Wen-Mei, W. H. (2016). Programming massively parallel processors: a hands-on approach. Morgan kaufmann.
- [3] NVIDIA: CUDA Zone (2022).
<https://developer.nvidia.com/cuda-zone>
- [4] NVIDIA: CUDA C Programming Guide (2022).
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- [5] NVIDIA: CUDA C Best Practices Guide (2022).
<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>
- [6] NVIDIA: GPU-Accelerated Applications (2022).
<https://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf>
- [7] Lin, D. L., & Huang, T. W. (2021). Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism. IEEE Transactions on Parallel and Distributed Systems, 33(11), 3041-3052.
- [8] Wang, Y., Feng, B., & Ding, Y. (2022, April). QGTC: accelerating quantized graph neural networks via GPU tensor core. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp. 107-119).
- [9] Zhi, X., Yan, J., Hang, Y., & Wang, S. (2019). Realization of CUDA-based real-time registration and target localization for high-resolution video images. Journal of Real-Time Image Processing, 16(4), 1025-1036.
- [10] Abed, S. E., Ali, M. H., & Al-Shayej, M. (2020). Enhanced GPU-based anti-noise hybrid edge detection method. COMPUTER SYSTEMS SCIENCE AND ENGINEERING, 35(1), 21-37.
- [11] Cavuoti, S., Garofalo, M., Brescia, M., Paolillo, M., Pescapo, A., Longo, G., & Ventre, G. (2014). Astrophysical data mining with GPU. A case study: genetic classification of globular clusters. New Astronomy, 26, 12-22.
- [12] Jurczuk, K., Czajkowski, M., & Kretowski, M. (2021). Multi-GPU approach to global induction of classification trees for large-scale data mining. Applied Intelligence, 51(8), 5683-5700.

- [13] Su, H., Zhang, C., Chai, J., Wen, M., Wu, N., & Ren, J. (2011, August). A high-efficient software parallel CAVCL encoder based on GPU. In 2011 34th International Conference on Telecommunications and Signal Processing (TSP) (pp. 534-540). IEEE.
- [14] Su, H., Wen, M., Wu, N., Ren, J., & Zhang, C. (2014). Efficient parallel video processing techniques on GPU: from framework to implementation. *The Scientific World Journal*, 2014.
- [15] Ahmad, A., Muharam, A., & Amira, A. (2017). GPU-based implementation of CABAC for 3-Dimensional Medical Image Compression. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 9(3-8), 45-50.
- [16] Balevic, A. (2009, August). Parallel variable-length encoding on GPGPUs. In *European Conference on Parallel Processing* (pp. 26-35). Springer, Berlin, Heidelberg.
- [17] Rahmani, H., Topal, C., & Akinlar, C. (2014, December). A parallel Huffman coder on the CUDA architecture. In *2014 IEEE Visual Communications and Image Processing Conference* (pp. 311-314). IEEE.
- [18] Vigil, B.M.L.P.: 2015, November. Accelerating the Canny edge detection algorithm with CUDA/GPU, *International Congress COMPUMAT (2015)*
- [19] Asan, M. A., & Ozsoy, A. (2021). cuRCD: Region covariance descriptor CUDA implementation. *Multimedia Tools and Applications*, 80(13), 19737-19751.
- [20] Wang, J., Zhang, X., Li, Y., & Lin, Y. (2021, August). Exploring HW/SW co-optimizations for accelerating large-scale texture identification on distributed GPUs. In *50th International Conference on Parallel Processing* (pp. 1-10).
- [21] Davy, A., & Ehret, T. (2021). GPU acceleration of NL-means, BM3D and VBM3D. *Journal of Real-Time Image Processing*, 18(1), 57-74.
- [22] Fuentes-Alventosa, A., Gómez-Luna, J., González-Linares, J. M., & Guil, N. (2014, October). CUVLE: Variable-length encoding on CUDA. In *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing* (pp. 1-6). IEEE.
- [23] Fuentes-Alventosa, A., Gómez-Luna, J., González-Linares, J. M., Guil, N., & Medina-Carnicer, R. (2022). CAVLCU: an efficient GPU-based implementation of CAVLC. *The Journal of Supercomputing*, 78(6), 7556-7590.
- [24] NVIDIA: NVENC Video Encoder API Programming Guide (2022).
<https://developer.nvidia.com/nvidia-video-codec-sdk>
- [25] Mian C, Jia J, Lei Y (2007) An H. 264 Video Encryption Algorithm Based on Entropy Coding. In: *Third International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP 2007)* (Vol. 2, pp. 41-44). IEEE

-
- [26] Tabash FK, Izharuddin M (2017) Efficient Encryption Technique for H. 264/AVC Based on CAVLC and Baker's Map. In: 2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI) (pp. 2759-2764). IEEE
- [27] Tabash FK, Izharuddin M, Tabash MI (2019) Encryption techniques for H. 264/AVC videos: a literature review. *J Inform Secur Appl* 45:20–34
- [28] Wang Y, O'Neill M, Kurugollu F (2013) A tunable encryption scheme and analysis of fast selective encryption for CAVLC and CABAC in H. 264/AVC. *IEEE Trans Circ Syst Video Technol* 23(9):1476–1490
- [29] Kim SM, Kim SB, Hong Y, Won CS (2007) Data Hiding on H. 264/AVC Compressed Video. In: *International Conference Image Analysis and Recognition* (pp. 698-707). Springer, Berlin, Heidelberg
- [30] Liao K, Lian S, Guo Z, Wang J (2012) Efficient information hiding in H 264/AVC video coding. *Telecommun Syst* 49(2):261–269
- [31] Xu D, Wang R, Shi YQ (2014) Data hiding in encrypted H. 264/AVC video streams by codeword substitution. *IEEE Trans Inform Foren Secur* 9(4):596–606
- [32] Xu D, Wang R, Shi YQ (2016) An improved scheme for data hiding in encrypted H. 264/AVC videos. *J Vis Commun Image Rep* 36:229–242
- [33] Mohanty M, Ooi W T (2012) Histopathology Image Streaming. In: *Pacific-Rim Conference on Multimedia* (pp. 534-545). Springer, Berlin, Heidelberg
- [34] Priya C, Ramya C (2018) Medical image compression based on fuzzy segmentation. *Int J Pure Appl Math* 118(20):603–610
- [35] Sridhar KV, Prasad KK (2008) Medical Image Compression Using Advanced Coding Technique. In: *2008 9th International Conference on Signal Processing* (pp. 2142-2145). IEEE
- [36] Fuentes-Alventosa, A., Gómez-Luna, J., & Medina-Carnicer, R. (2022). GUD-Canny: A real-time GPU-based unsupervised and distributed Canny edge detector. *Journal of Real-Time Image Processing*, 19(3), 591-605.
- [37] Luo, Y., Duraiswami, R.: Canny edge detection on NVIDIA CUDA. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops* (pp. 1-8). IEEE (2008, June)
- [38] Ogawa, K., Ito, Y., Nakano, K.: Efficient Canny edge detection using a GPU. In *2010 First International Conference on Networking and Computing* (pp. 279-280). IEEE (2010, November)

[39] Palomar, R., Palomares, J.M., Castillo, J.M., Olivares, J., Gómez-Luna, J.: Parallelizing and optimizing lip-canny using nvidia cuda. In International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (pp. 389-398). Springer, Berlin, Heidelberg (2010, June)

[40] Lourenço, L. H., Weingaertner, D., Todt, E.: Efficient implementation of canny edge detection filter for ITK using CUDA. In 2012 13th Symposium on Computer Systems (pp. 33-40). IEEE (2012, October)

[41] Huang, Y., Bai, Y., Li, R., Huang, X.: Research of Canny edge detection algorithm on embedded CPU and GPU heterogeneous systems. In 2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNCFSKD) (pp. 647-651). IEEE (2016, August)

[42] Mogale, H.: High Performance Canny Edge Detector using Parallel Patterns for Scalability on Modern Multicore Processors. (2017) arXiv preprint arXiv: 1710. 07745

[43] Emrani, Z., Bateni, S., Rabbani, H.: A new parallel approach for accelerating the gpu-based execution of edge detection algorithms. *J. Med. Signals Sens.* 7(1), 33 (2017)

[44] Medina-Carnicer, R., Munoz-Salinas, R., Yeguas-Bolivar, E., & Diaz-Mas, L. (2011). A novel method to look for the hysteresis thresholds for the Canny edge detector. *Pattern Recognition*, 44(6), 1201-1211.

[45] Rao, D.V., Venkatesan, M.: An efficient reconfigurable architecture and implementation of edge detection algorithm using Handle-C. In International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. (Vol. 2, pp. 843-847). IEEE (2004, April)

[46] Neoh, H.S., Hazanchuk, A.: Adaptive edge detection for real-time video processing using FPGAs. *Global Signal Process.* 7(3), 2–3 (2004)

[47] Gentsos, C., Sotiropoulou, C.L., Nikolaidis, S., Vassiliadis, N.: Real-time canny edge detection parallel implementation for FPGAs. In 2010 17th IEEE International Conference on Electronics, Circuits and Systems (pp. 499-502). IEEE (2010, December)

[48] He, W., Yuan, K.: An improved Canny edge detector and its realization on FPGA. In 2008 7th World Congress on Intelligent Control and Automation (pp. 6561-6564). IEEE (2008, June)

[49] Li, X., Jiang, J., Fan, Q.: An improved real-time hardware architecture for Canny edge detection based on FPGA. In 2012 Third International Conference on Intelligent Control and Information Processing (pp. 445-449). IEEE (2012, July)

[50] Peng, F., Lu, X., Lu, H., Shen, S.: An improved high-speed canny edge detection algorithm and its implementation on FPGA. In Fourth International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition

and Basic Technologies (Vol. 8350, p. 83501V). International Society for Optics and Photonics (2012, January)

[51] Abdelgawad, H.M., Safar, M., Wahba, A.M.: High level synthesis of canny edge detection algorithm on Zynq platform. *Int. J. Comput. Electr. Autom. Control Inf. Eng* 9(1), 148–152 (2015)

[52] Xu, Q., Varadarajan, S., Chakrabarti, C., Karam, L.J.: A distributed canny edge detector: algorithm and FPGA implementation. *IEEE Trans. Image Process.* 23(7), 2944–2960 (2014)

[53] Sangeetha, D., Deepa, P.: FPGA implementation of cost-effective robust Canny edge detection algorithm. *J. Real-Time Image Proc.* 16(4), 957–970 (2019)

[54] Fuentes-Alventosa, A., Gómez-Luna, J., & Medina-Carnicer, R. (2022). GVLE: a highly optimized GPU-based implementation of variable-length encoding. *The Journal of Supercomputing*, 1-28.

[55] Yamamoto, N., Nakano, K., Ito, Y., Takafuji, D., Kasagi, A., & Tabaru, T. (2020, August). Huffman coding with gap arrays for GPU acceleration. In *49th International Conference on Parallel Processing-ICPP* (pp. 1-11).

[56] Luna, J. G. G., Chang, L. W., Sung, I. J., Hwu, W. M., & Guil, N. (2015, September). In-place data sliding algorithms for many-core architectures. In *2015 44th International Conference on Parallel Processing* (pp. 210-219). IEEE.

[57] Jayasankar, U., Thirumal, V., & Ponnurangam, D. (2021). A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University-Computer and Information Sciences*, 33(2), 119-140.

[58] Wise, J., "How many videos are uploaded to youtube a day in 2023?", November 2022.

<https://earthweb.com/how-many-videos-are-uploaded-to-youtube-a-day/>