GPU ACCELERATED AES

IMPLEMENTACIÓN EN GPU DEL CIFRADOR AES

ETSIINF DE LA UNIVERSIDAD POLITÉCNICA DE MADRID

RESEARCH CENTER FOR COMPUTATIONAL SIMULATION

RESEARCH GROUP ON QUANTUM INFORMATION AND COMPUTATION

AUTOR: JESÚS MARTÍN BERLANGA

TUTOR: JESÚS MARTÍNEZ MATEO

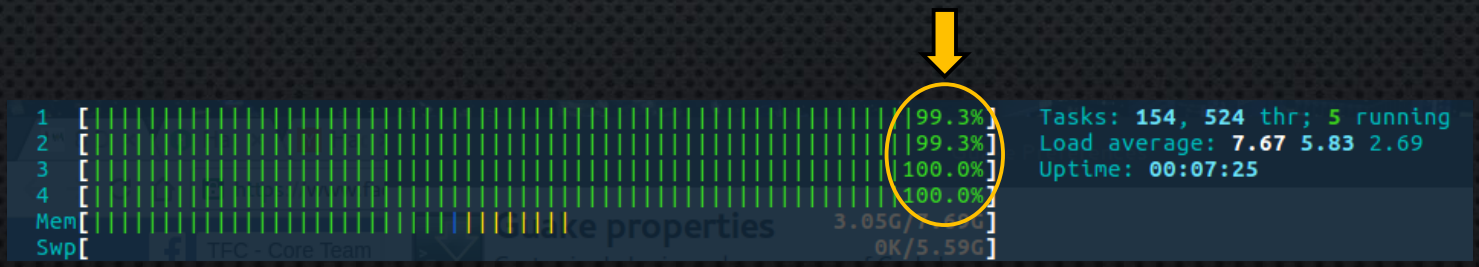
0. CONTENIDOS

1. Motivaciones
2. Objetivos
3. Fundamentos
4. Implementación y resultados
5. Comparación con versión CPU
6. Conclusiones

1. MOTIVACIONES

¿Por qué usar una GPU para cifrar/descrifrar?

- Uso en Sistemas Especializados
  - Cuellos de botella en el cifrador
  - Necesidad de nuevas soluciones que puedan sostener altas transferencias de datos
- Además, uso en computación "doméstica":
  - Aprovechar la GPUs potentes a cada vez precios mas asequibles para el usuario medio para
    - Evitar saturar la CPU al cifrar grandes ficheros
    - Poder seguir usando la CPU para otras tareas
    - Operación en menor tiempo



2. OBJETIVOS

- Realizar implementación CUDA del cifrador AES y revisando distintos factores de la implementación que pueden afectar al rendimiento

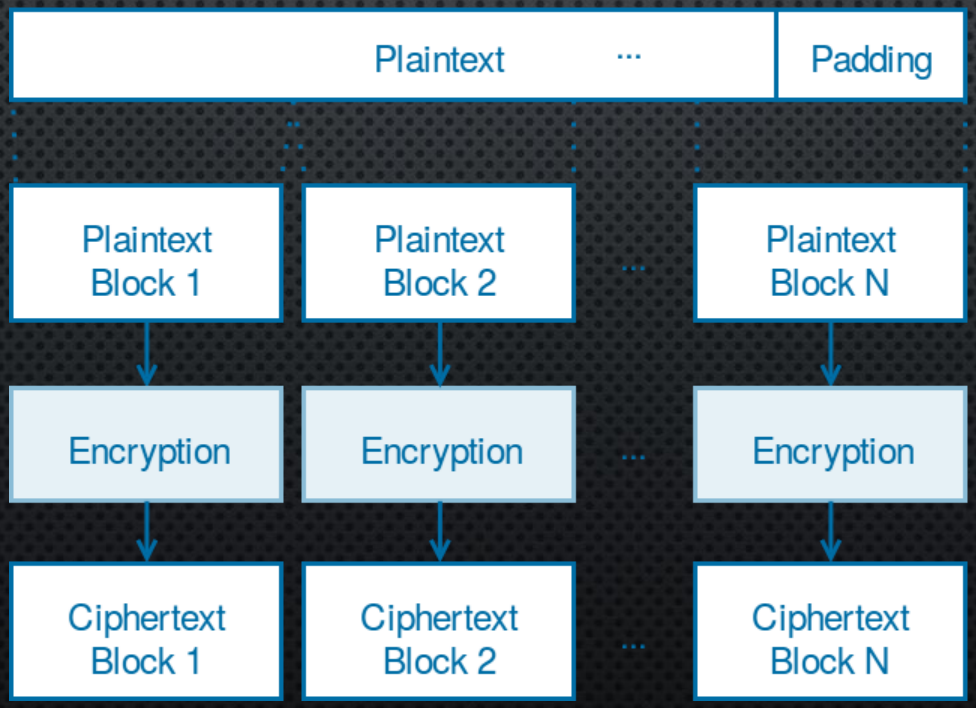
- Descubrir limitaciones de la versión GPU en contraste con una CPU para explorar su viabilidad en Sistemas Especializados
  - Pensado usarse con distribución de claves cuánticas de alto rendimiento

3. FUNDAMENTOS

3.1 Cifrado por bloque
3.2 GPU
3.3 Cifrado en GPU
3.4 Cifrado AES
3.5 Modos de operación

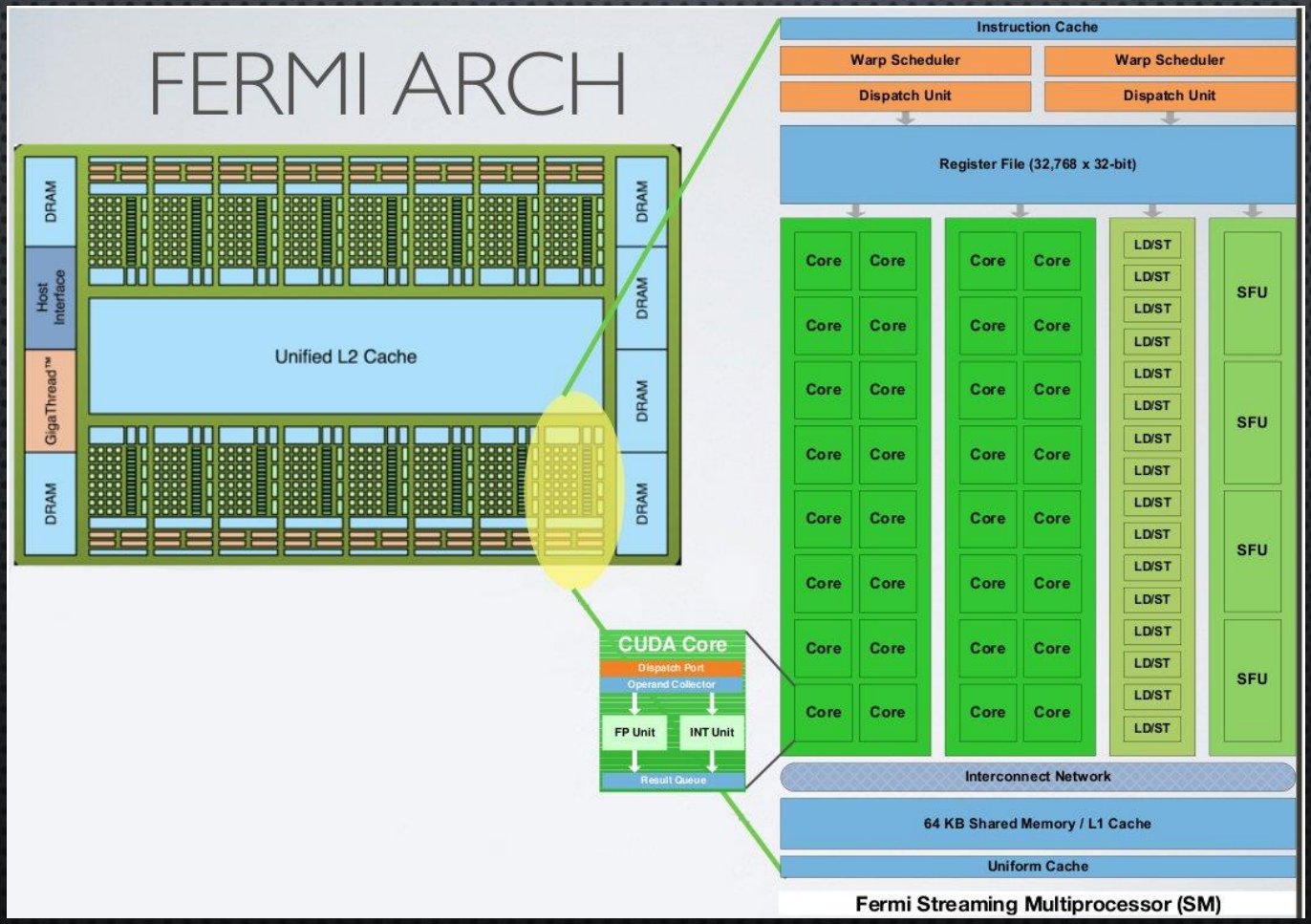
3.1 CIFRADO POR BLOQUE

- Se opera con grupos de bits de longitud fija (bloques)
- La misma operación de cifrado (cifrado o descrifrado) con la misma clave se aplica a cada bloque



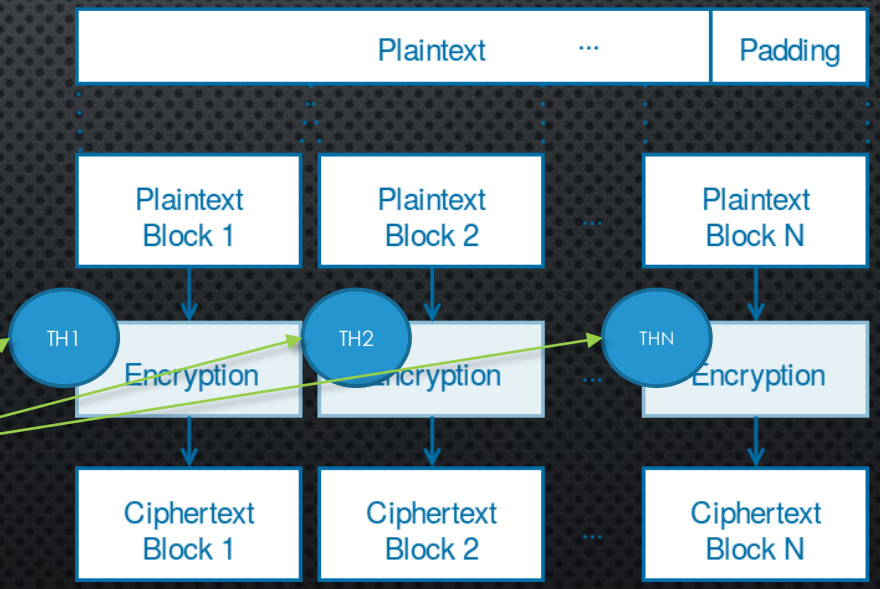
3.2 GPU

- Una GPU como la GTX 780 tiene 2304 CUDA Cores agrupados en multiprocesadores cada uno con sus registros y caché.



3.3 CIFRADO EN GPU

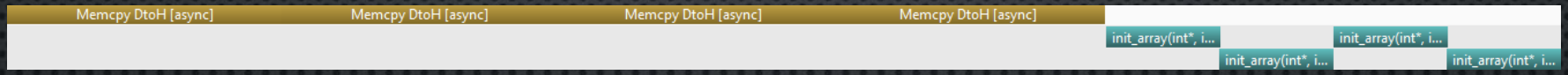
- Como cada bloque se puede cifrar de forma independiente a los otros, la idea es que al cifrarlos con 2304 Cores de la GPU en lugar de solo 8 de la CPU podremos obtener un gran beneficio.



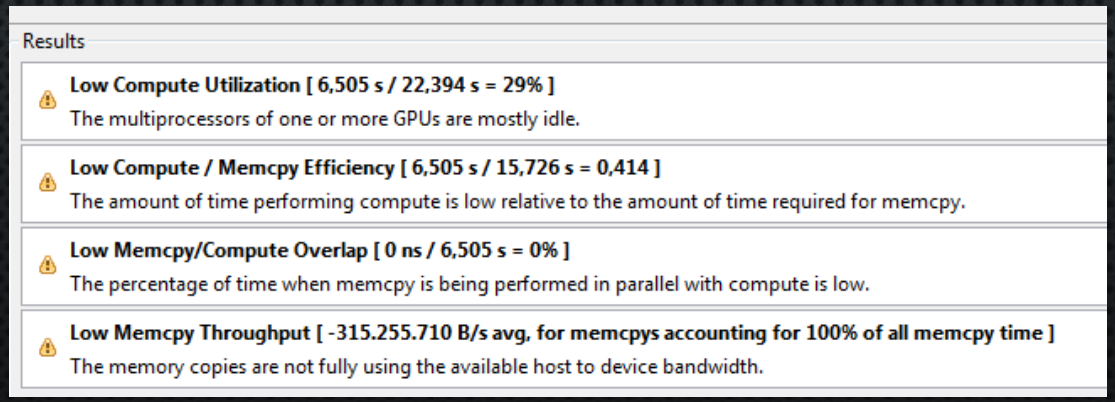
Cifrar cada bloque Independiente con un thread distinto en paralelo

3.3 CIFRADO EN GPU

- A tener en cuenta:
  - Un Core en la GPU es menos potente y mucho mas simple que un Core de la CPU
  - Las transferencias de memoria son caras - con pocos datos el tiempo de transferencia puede ser mayor que el tiempo de computo en la CPU

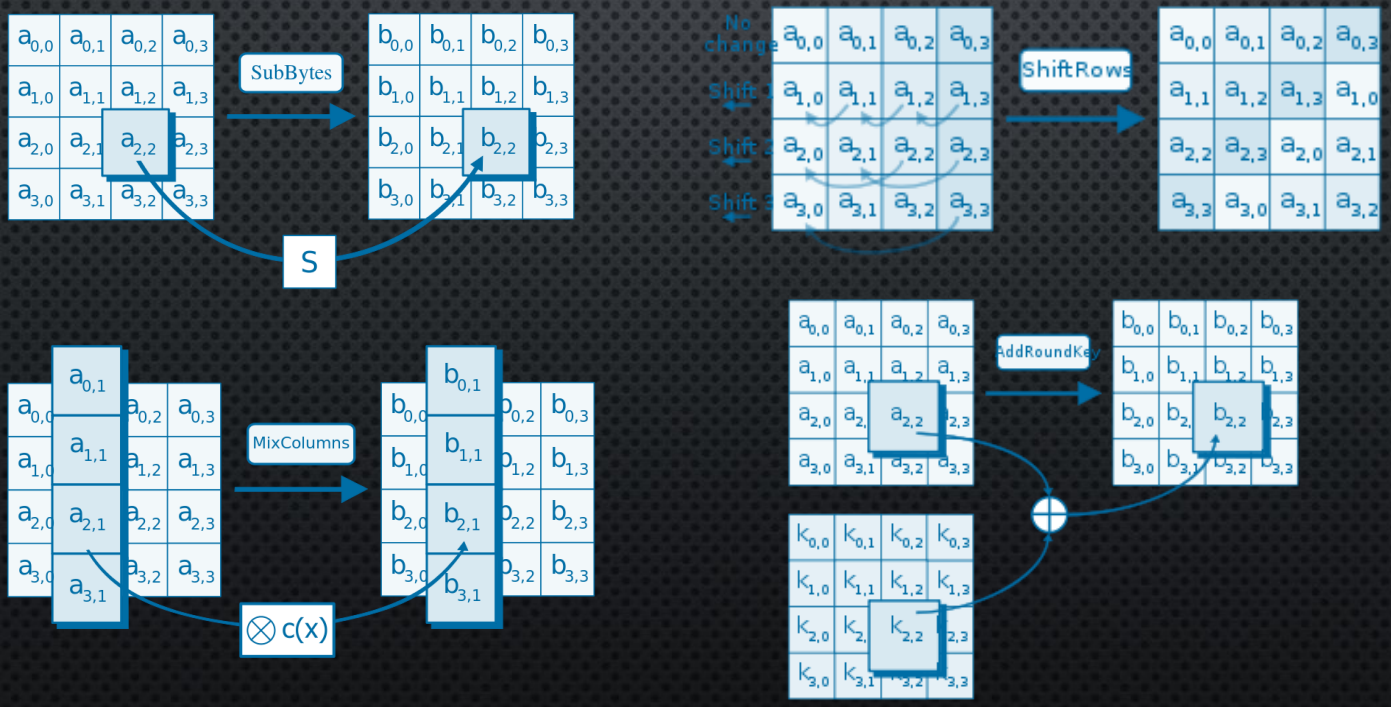


  - Se necesitan muchos datos (ficheros grandes) para aprovechar el hardware GPU y notar beneficios



3.4 CIFRADO AES

- Bloques de 16 bytes (128 bits)
- Cada bloque de entrada sufre una serie de transformaciones repetidas durante varias rondas



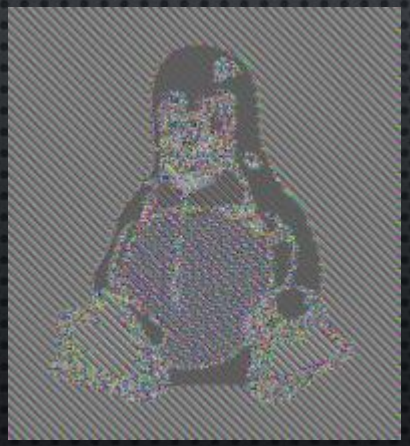
3.4 CIFRADO AES

- Al expandir la clave original se obtienen las claves de ronda con tamaño siempre de128 bits
- El algoritmo de expansión es estrictamente secuencial y se ha reutilizado el código de OpenSSL

3.5 MODOS DE OPERACIÓN

- Modos de operación necesarios para un cifrado seguro

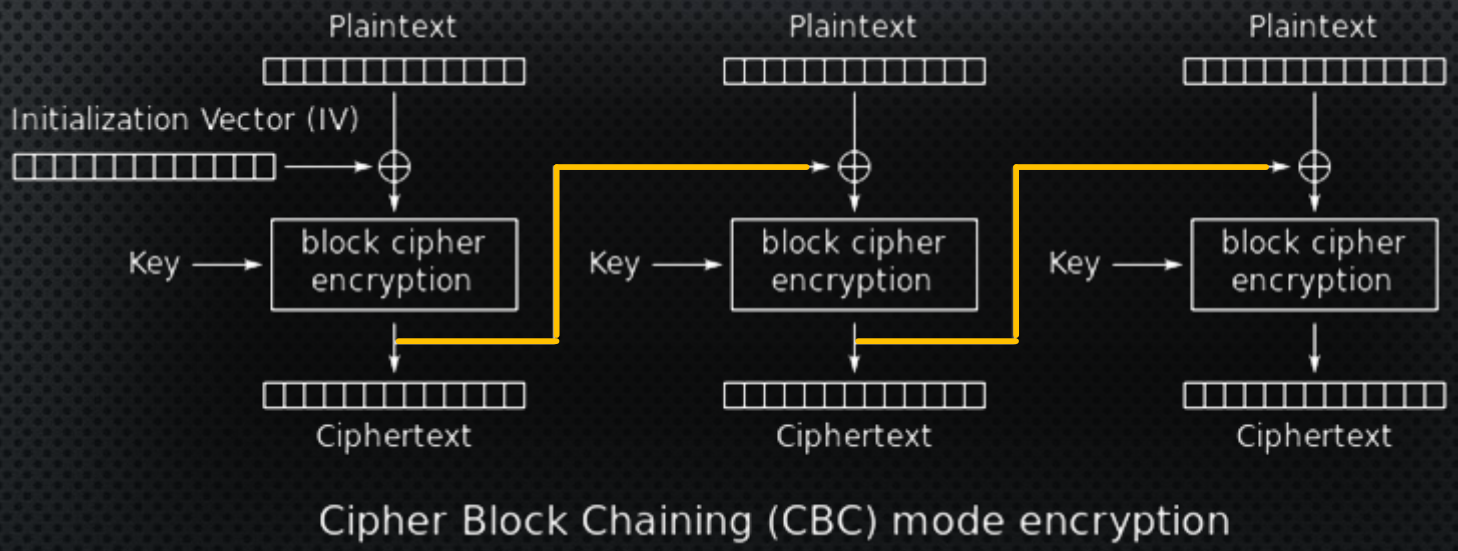
Si ciframos tal como hemos explicado (modo ECB):



Con otros modos de operación:

3.5 MODOS DE OPERACIÓN

Problema: No todos los modos de operación son paralelizables



Cipher Block Chaining (CBC) mode encryption

Dependencias entre bloques: para cifrar el siguiente
Necesitamos haber cifrado el anterior

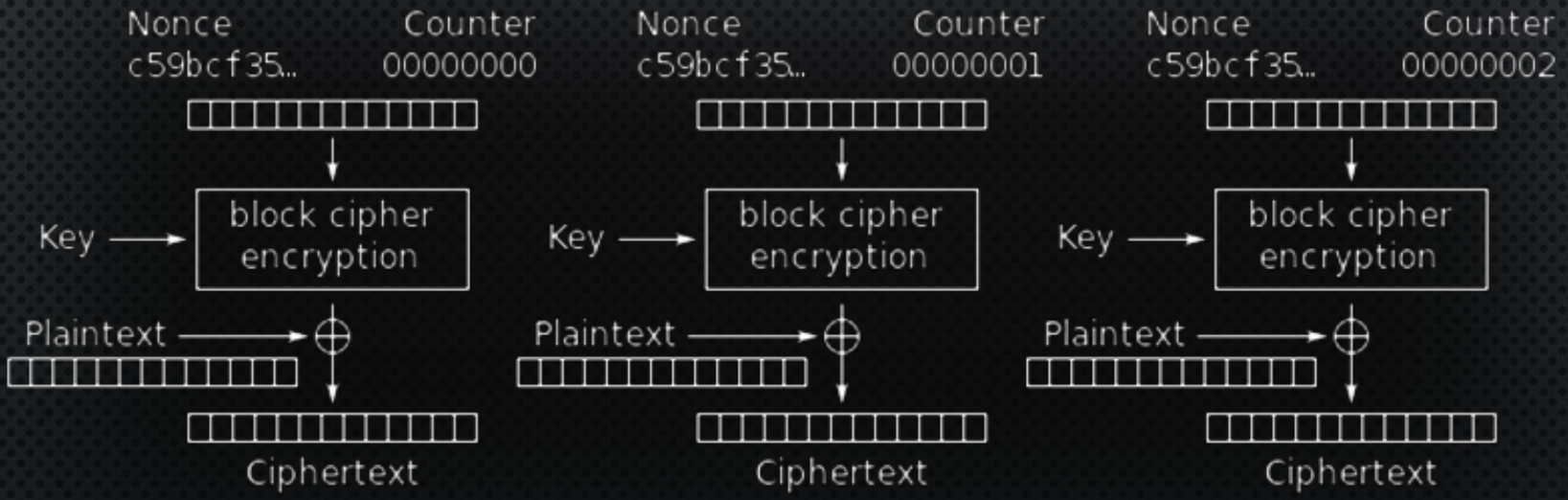
3.5 MODOS DE OPERACIÓN

A parte del modo básico ECB no seguro se ha implementado el modo contador (CTR), y los modos CBC y CFB solo para descifrado.

Modo	Cifrado paralelizable	Descifrado paralelizable
ECB	SI	SI
CBC	NO	**SI**
PCBC	NO	NO
CFB	NO	**SI**
OFB	NO	NO
CTR	**SI**	**SI**

3.5 MODOS DE OPERACIÓN

Modo contador (CTR)



Counter (CTR) mode encryption

4. IMPLEMENTACIÓN Y RESULTADOS

4.1 Organización de los threads
4.2 Nivel de paralelismo
4.3 Tablas de búsqueda
4.4 Acceso bytes estado
4.5 Transferencias host-gpu
4.6 Acceso aleatorio

4.1 ORGANIZACIÓN DE LOS THREADS

- Se establece un tamaño de bloque (de threads) que maximize la ocupancia, es decir, que todos los threads y bloques de trheads disponibles por multiprocesador se utilizen

	Compute Capability										
Technical Specifications	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
Maximum number of concurrent kernels	16		4	32				16	128	32	16
Maximum number of threads per block	1024										
Max. resident threads per multiprocessor	1536	2048									
Max. resident warps per multiprocessor	48	64									
Warp size	32										
Max. resident blocks per multiprocessor	8	16				32					

Algunos datos de la Tabla 14 del Apéndice G
del manual de programación CUDA de NVIDIA

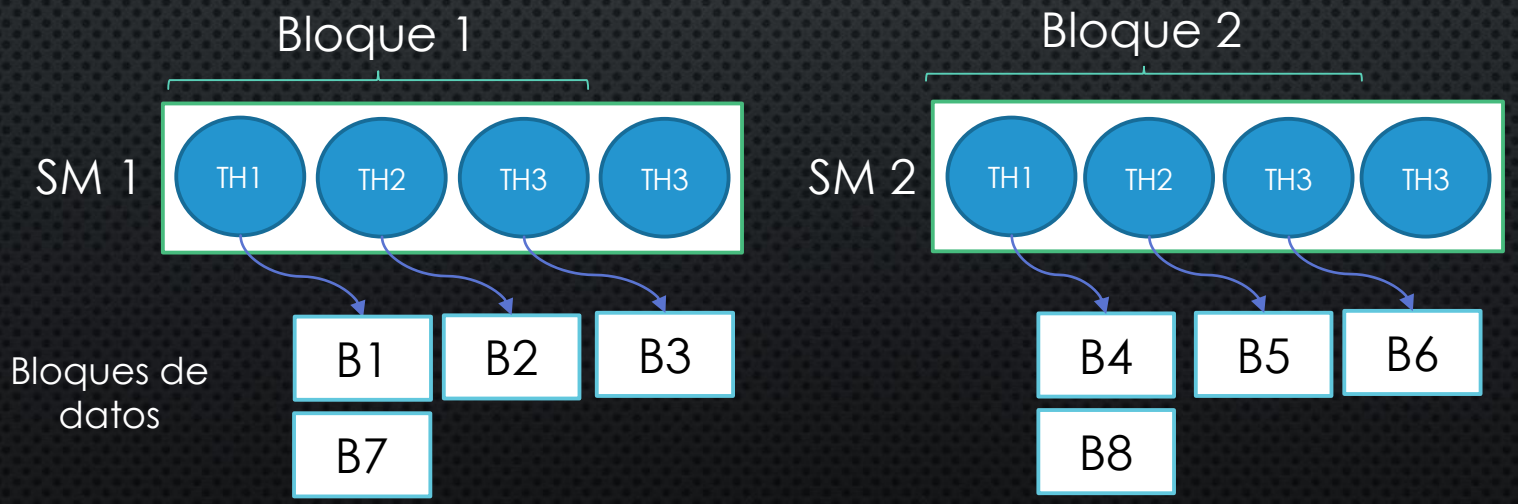
4.1 ORGANIZACIÓN DE LOS THREADS

Ejemplo sencillo:

- GPU con 2 multiprocesadores (SM)
- Cada SM tiene una serie de soporta un máximo de 4 threads y 2 bloques de threads

Si fijamos el tamaño de bloque en 3 *threads* se utilizan:

- 2/2 bloques por multiprocesador → 100%
- 3/4 threads por multiplocesador → 75%



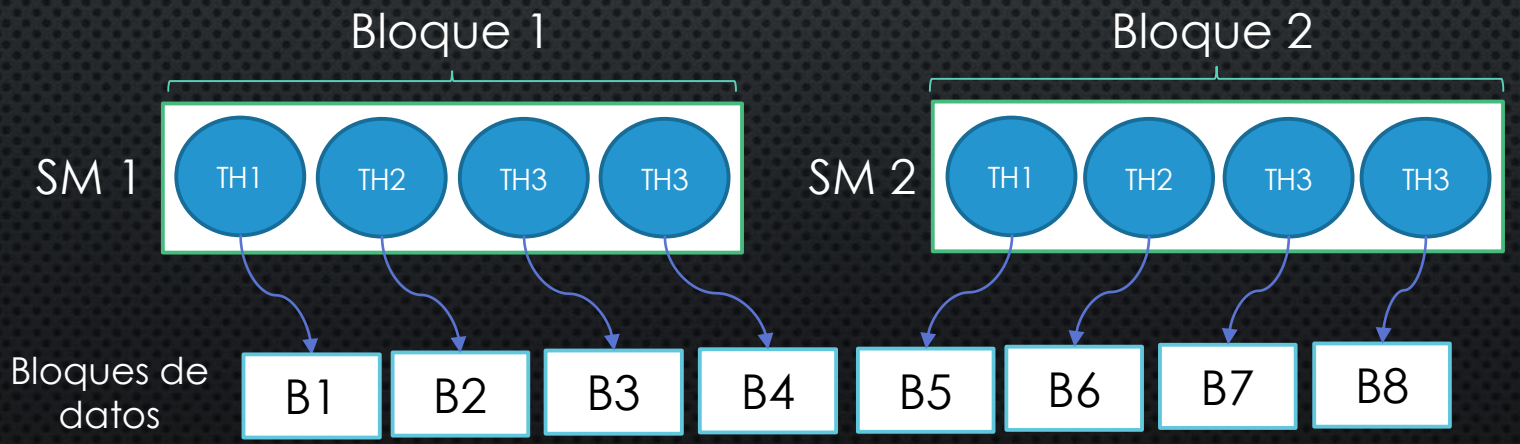
4.1 ORGANIZACIÓN DE LOS THREADS

Ejemplo sencillo:

- GPU con 2 multiprocesadores (SM)
- Cada SM tiene una serie de soporta un máximo de 4 threads y 2 bloques de threads

Si fijamos el tamaño de bloque en 4 *threads* se utilizan:

- 1/2 bloques por multiprocesador → 50%
- 4/4 threads por multiplocesador → 100%

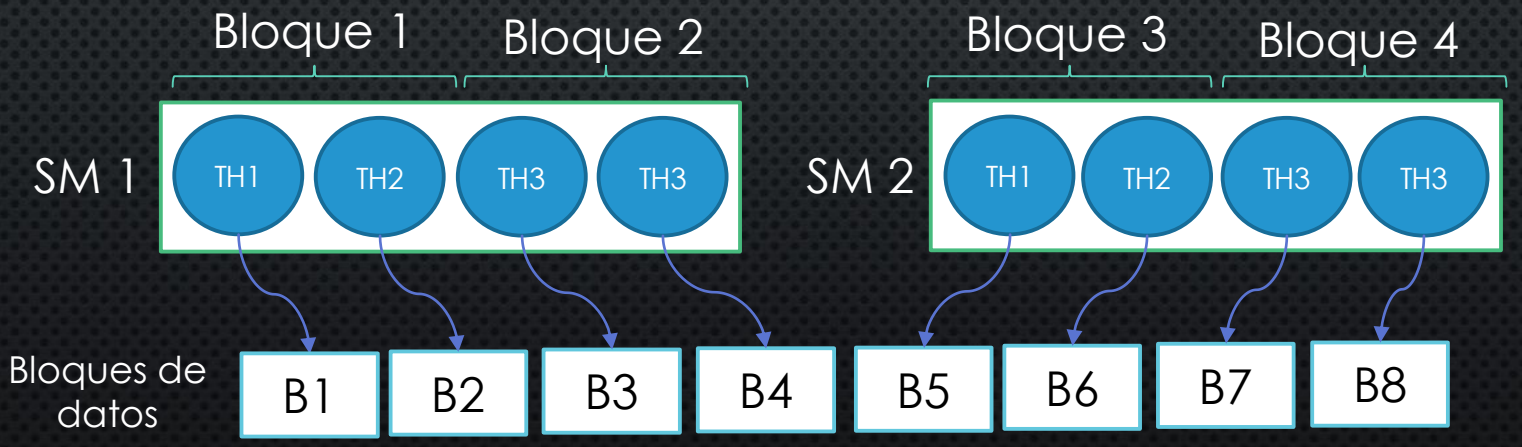


4.1 ORGANIZACIÓN DE LOS THREADS

$$\text{threads por bloque de threads} = \frac{\text{max. threads residentes por SM}}{\text{max. bloques residentes por SM}}$$

Si fijamos el tamaño de bloque en 4/2 = 2 *threads* se utilizan:

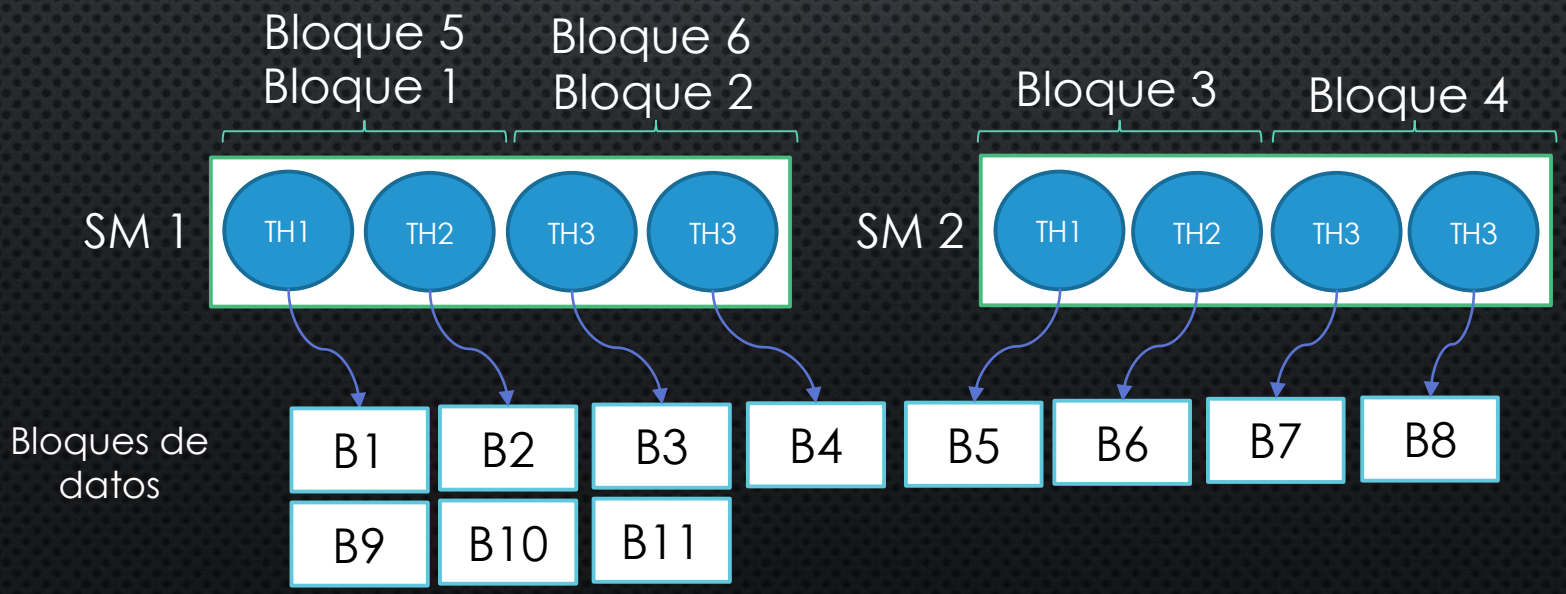
- 2/2 bloques por multiprocesador → 100%
- 4/4 threads por multiplocesador → 100%



4.1 ORGANIZACIÓN DE LOS THREADS

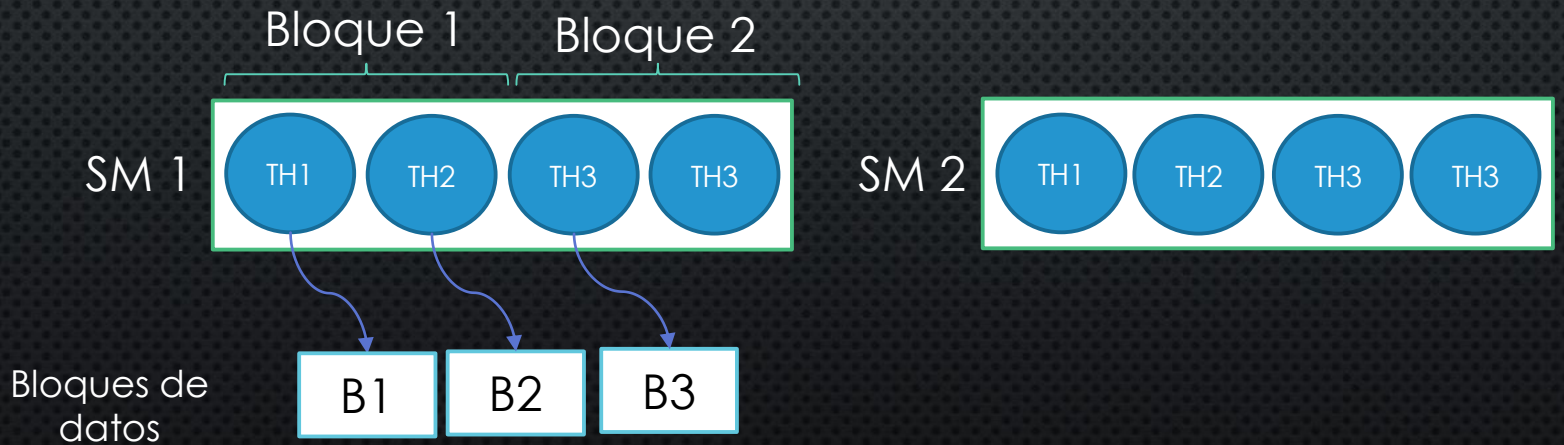
- Una vez calculado el tamaño de bloque, se crean suficientes bloques de *threads* para procesar todos los bloques de datos que se quieren cifrar/descrifrar

Ej.: Para un cifrar 11 bloques de texto se crean $\lceil 11/2 \rceil = 6$ bloques de *threads*



4.1 ORGANIZACIÓN DE LOS THREADS

- Si no hay suficientes bloques de datos no utilizaremos todo el poder de la GPU



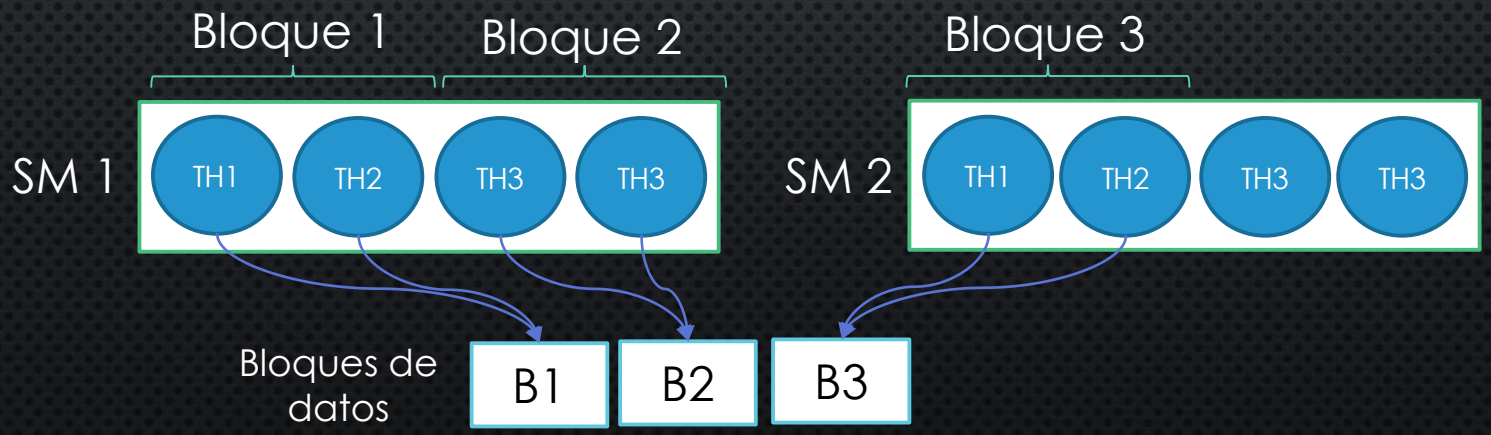
4.2 NIVEL DE PARALELISMO

- Se han implementado versiones que utilizan mas de un thread (dos, cuatro, y dieciséis) para procesar cada bloque de datos

- Para 1 thread por bloque de datos: $\lceil 3/2 \rceil = 2$ bloques
- Para 2 threads por bloque de datos: $\lceil 6/2 \rceil = 3$ bloques
- Para 4 threads por bloque de datos: $\lceil 12/2 \rceil = 6$ bloques
- Para 16 threads por bloque de datos: $\lceil 48/2 \rceil = 24$ bloques

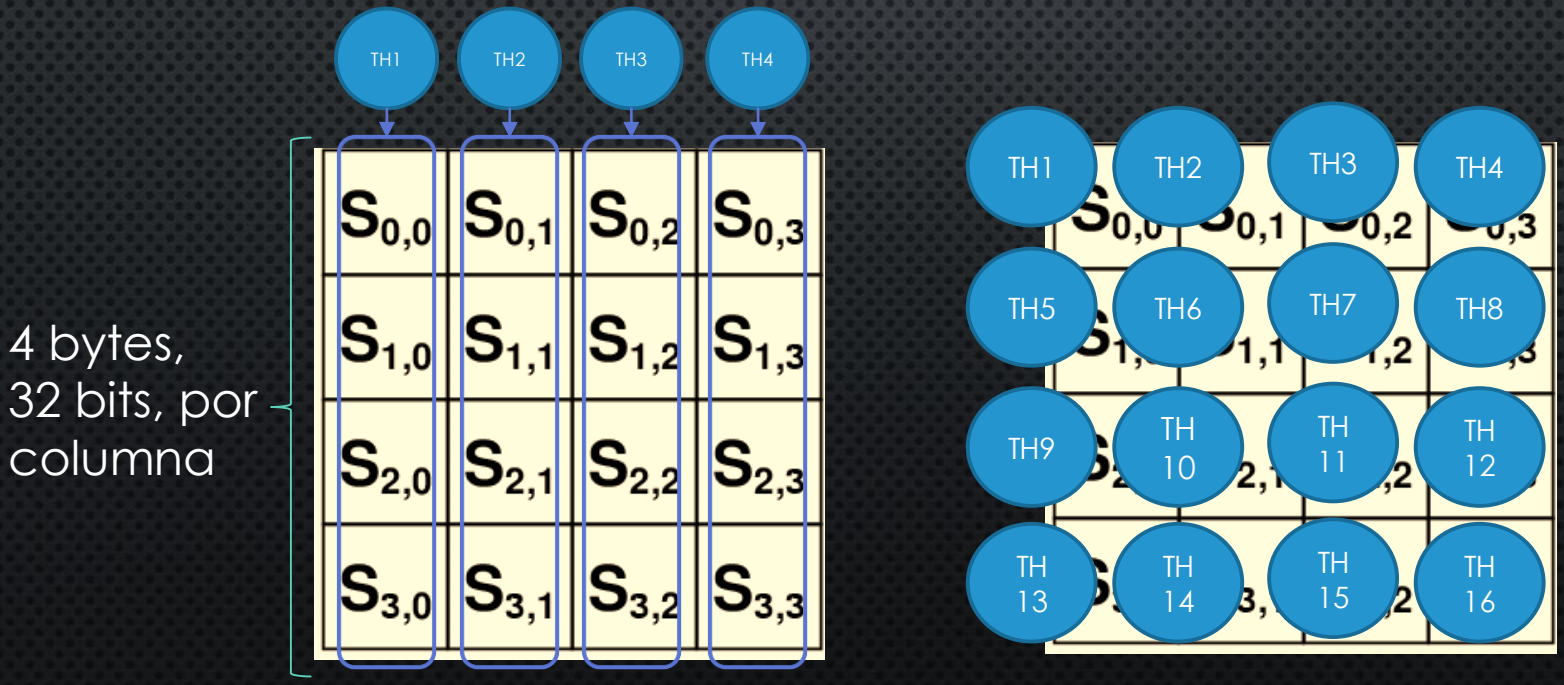
+ ocupancia

Ejemplo 2 threads por bloque de datos:



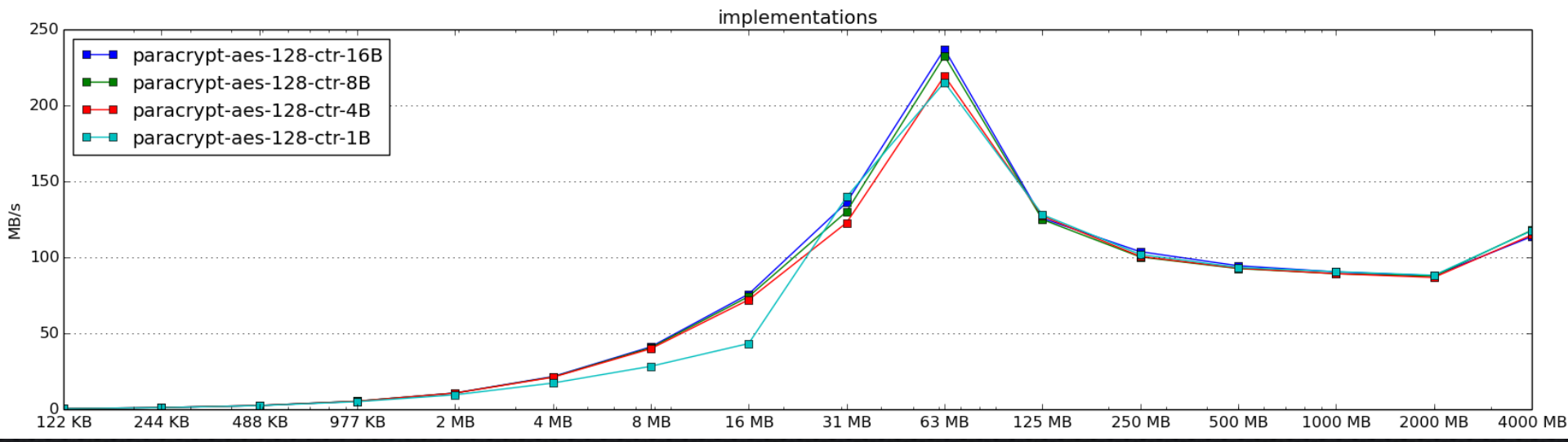
4.2 NIVEL DE PARALELISMO

- Al tener que trabajar varios threads con el mismo conjunto de datos es necesario incluir mecanismos de sincronización
  → más problemático
- Además, con ficheros grandes no hay problema de ocupancia.
- Con 16 threads no se aprovechan los registros de 32 bits de la GPU



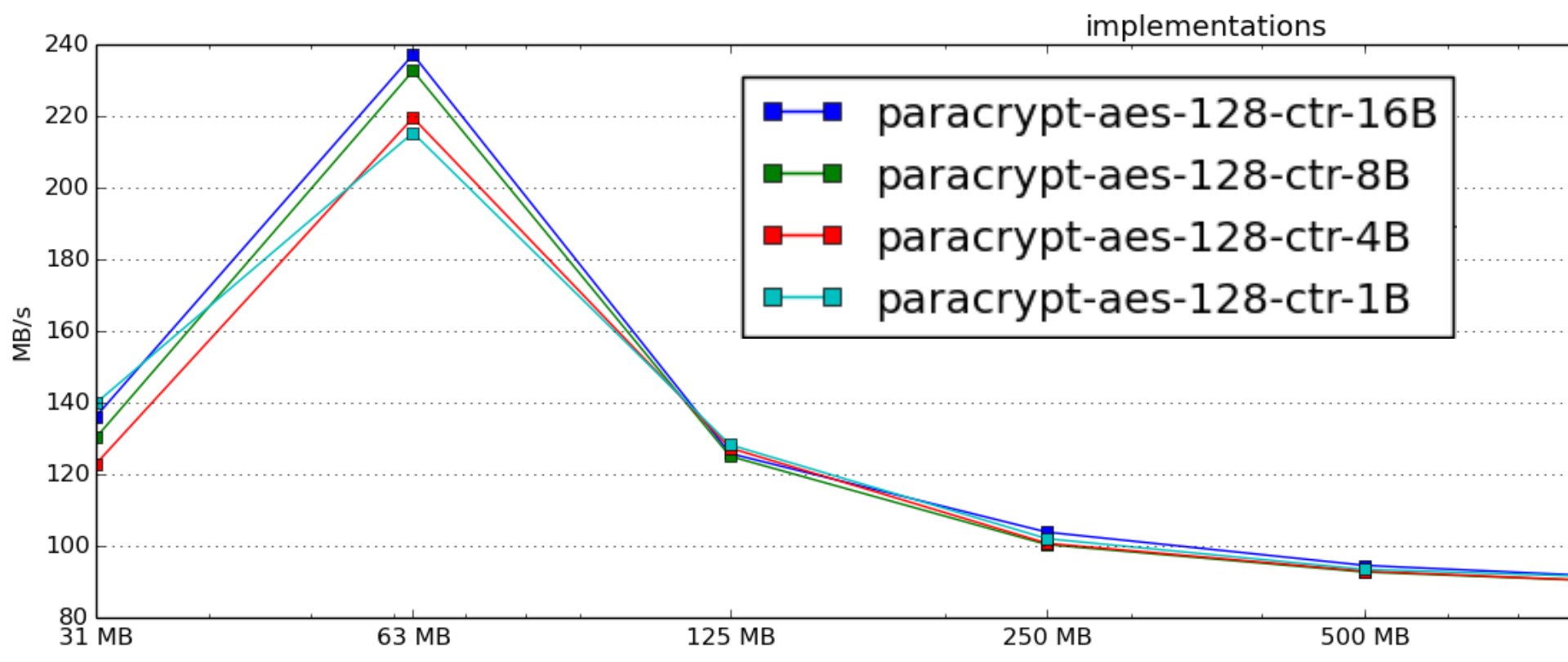
4.2 NIVEL DE PARALELISMO

- Se esperaban mayores diferencias resultados
- Es posible que un cuello de botella en el disco no haya permitido obtener mayores diferencias conforme el tamaño de fichero aumenta



4.2 NIVEL DE PARALELISMO

- Se esperaban mayores diferencias resultados
- Es posible que un cuello de botella en el disco no haya permitido obtener mayores diferencias conforme el tamaño de fichero aumenta



4.3 TABLAS DE BÚSQUEDA

Ya sabemos como dividir el trabajo en bloques de *threads* pero, ¿Qué realiza exactamente cada *thread* con los bloques de datos que le toca procesar?

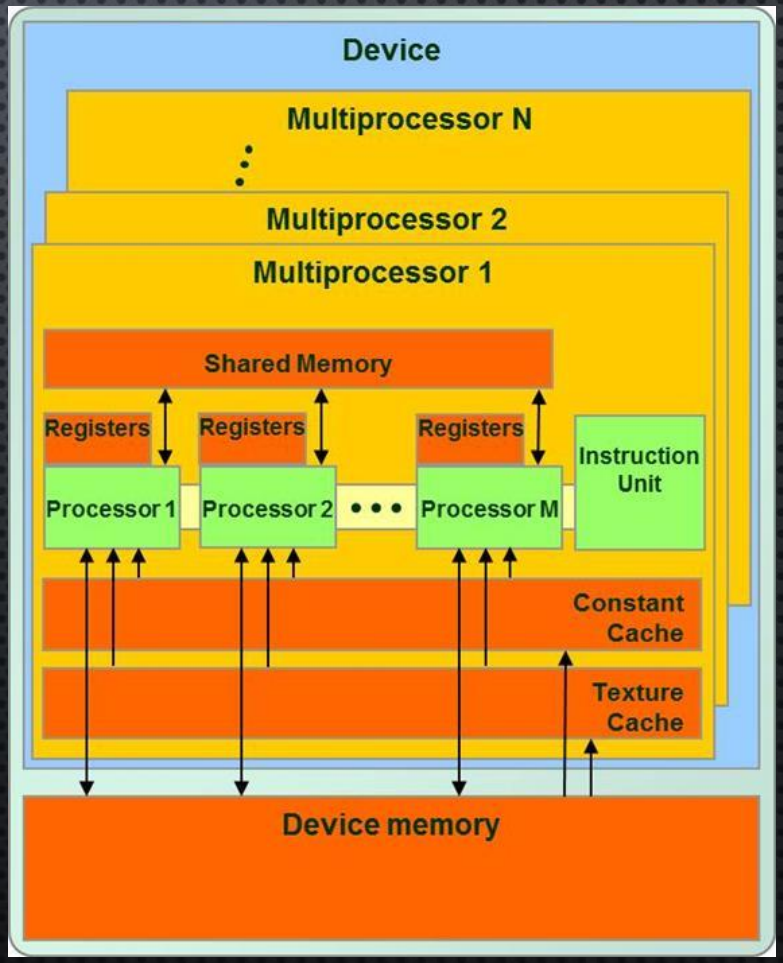
- En lugar de realizar las operaciones de SubBytes, ShiftRows y MixColumns en cada ronda se usa resultado precomputado en una tabla (extraída del código fuente de OpenSSL) lo que reduce significamente el numero de operaciones necesarias y aumenta el rendimiento

$$\begin{matrix} S_0 & S_1 & S_2 & S_3 \end{matrix}$$
$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$S'_0 = T_0[a_{00}] \oplus T_1[a_{11}] \oplus T_2[a_{22}] \oplus T_3[a_{33}] \oplus k_0$$
$$S'_1 = T_0[a_{01}] \oplus T_1[a_{12}] \oplus T_2[a_{23}] \oplus T_3[a_{30}] \oplus k_1$$
$$S'_2 = T_0[a_{02}] \oplus T_1[a_{13}] \oplus T_2[a_{20}] \oplus T_3[a_{31}] \oplus k_2$$
$$S'_3 = T_0[a_{03}] \oplus T_1[a_{10}] \oplus T_2[a_{21}] \oplus T_3[a_{32}] \oplus k_3$$

4.3 TABLAS DE BÚSQUEDA

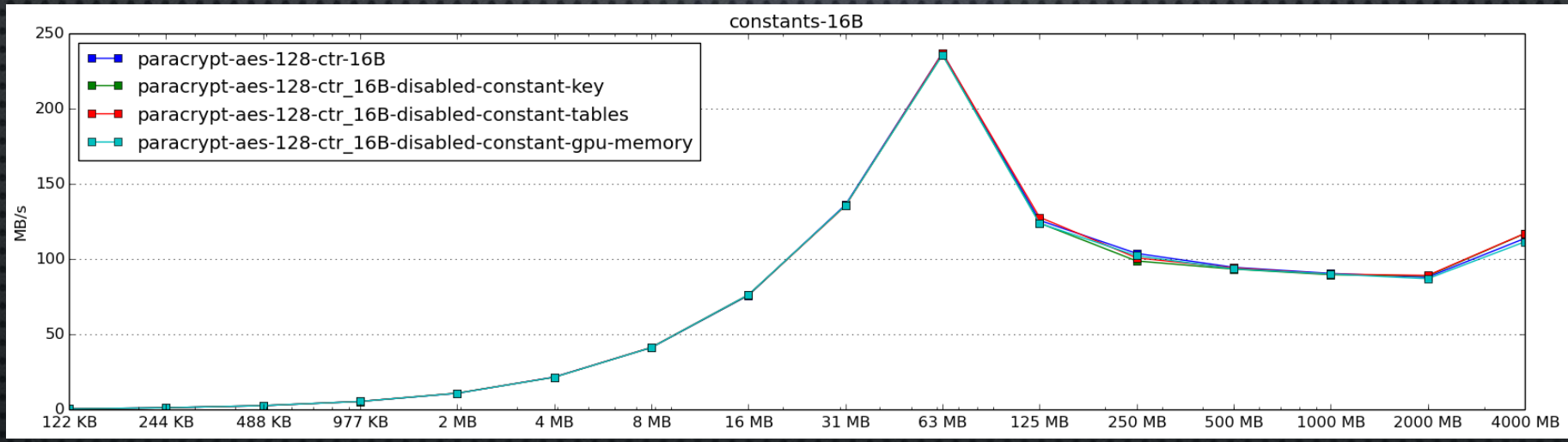
- Tanto las tablas de búsqueda como las claves de ronda pueden almacenarse en memoria constante porque son de solo lectura
- Memoria constante esta pensada para hacer "broadcast" y funciona mejor cuando varios *threads* acceden a la misma posición… como pasa con las claves de ronda
- La memoria compartida es una cache L1, podemos guardar manualmente las tablas de búsqueda en memoria compartida



4.3 TABLAS DE BÚSQUEDA

Implementación con tablas de búsqueda

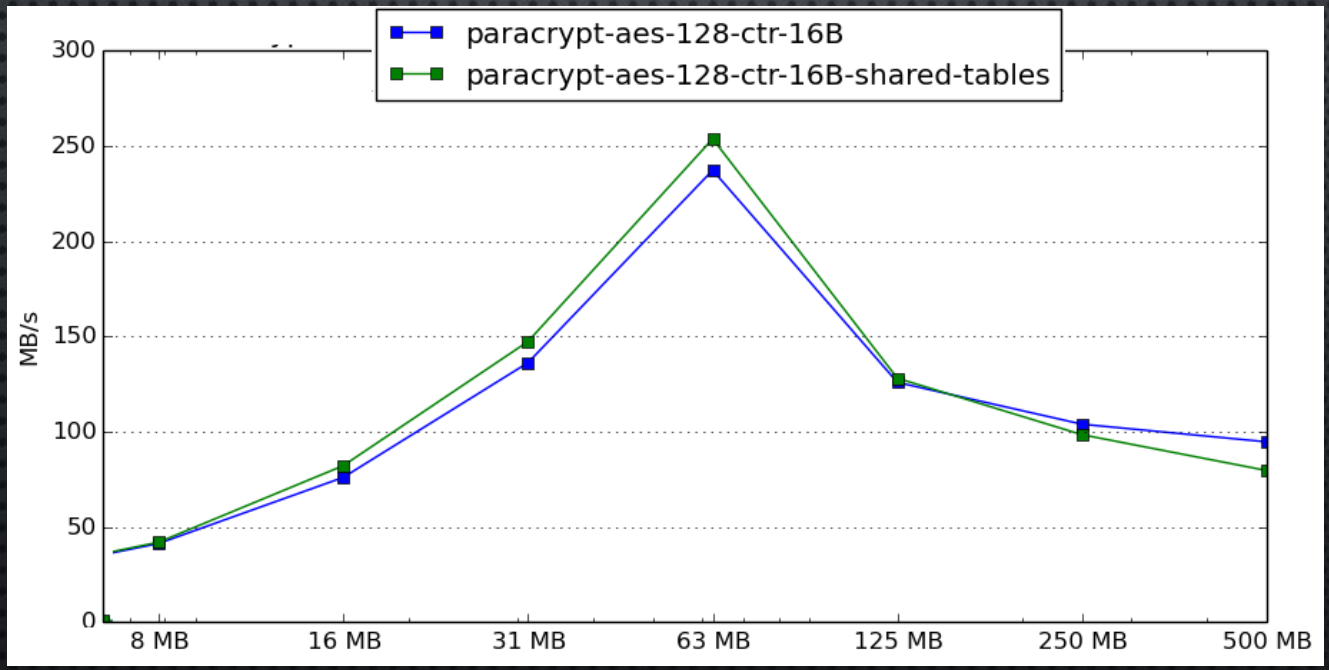
- Apenas diferencias con el uso de memoria constante



4.3 TABLAS DE BÚSQUEDA

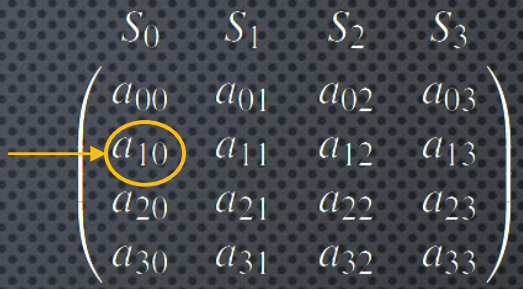
Implementación con tablas de búsqueda

Impacto de almacenar manualmente tablas de búsqueda en memoria compartida (cache):



4.4 ACCESO BYTES ESTADO

Operaciones con enteros



- Operaciones con enteros para acceder bytes de una palabra optimizadas en CPU

```
uint32_t s0;
uint8_t a10 = (s0 >> 8) & 0xff;
```

- En la GPU en cambio son de las operaciones más caras

	Compute Capability								
Arithmetic instruction & throughput	2.0	2.1	3.0, 3.2	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2
32-bit floating-point add, multiply, multiply-add	32	48	192	192	128	128	64	128	128
32-bit integer shift	16	16	32	64	64	64	32	64	64
32-bit bitwise AND, OR, XOR	32	48	160	160	128	128	64	128	128

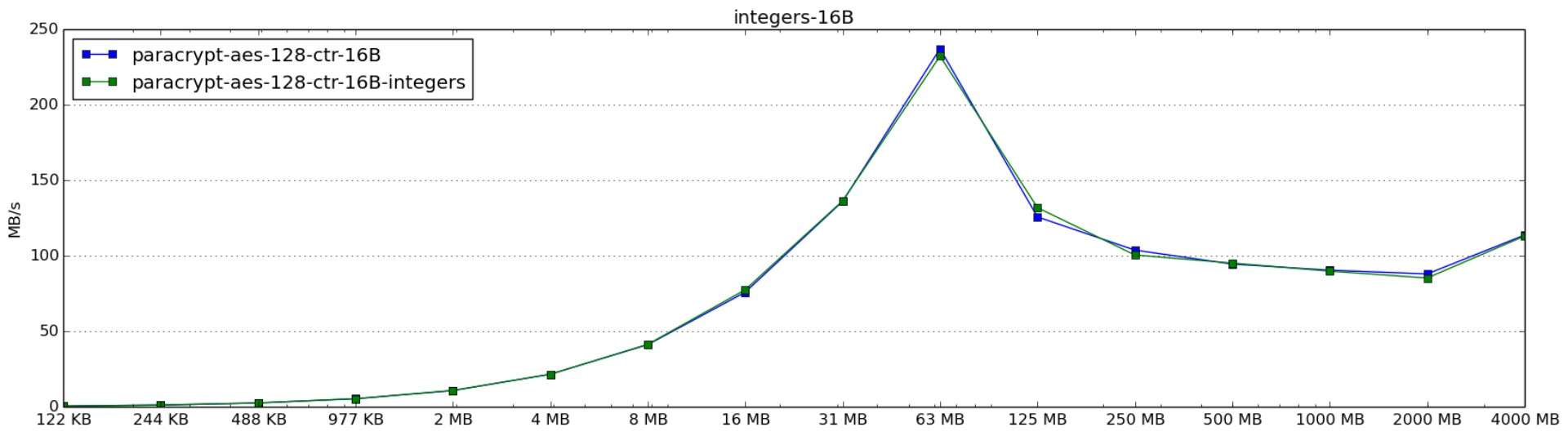
→ alternativa: uso de punteros en GPU

```
uint8_t* s0p = (uint8_t*) &s0;
uint8_t a10 = s0p[1];
```

4.4 ACCESO BYTES ESTADO

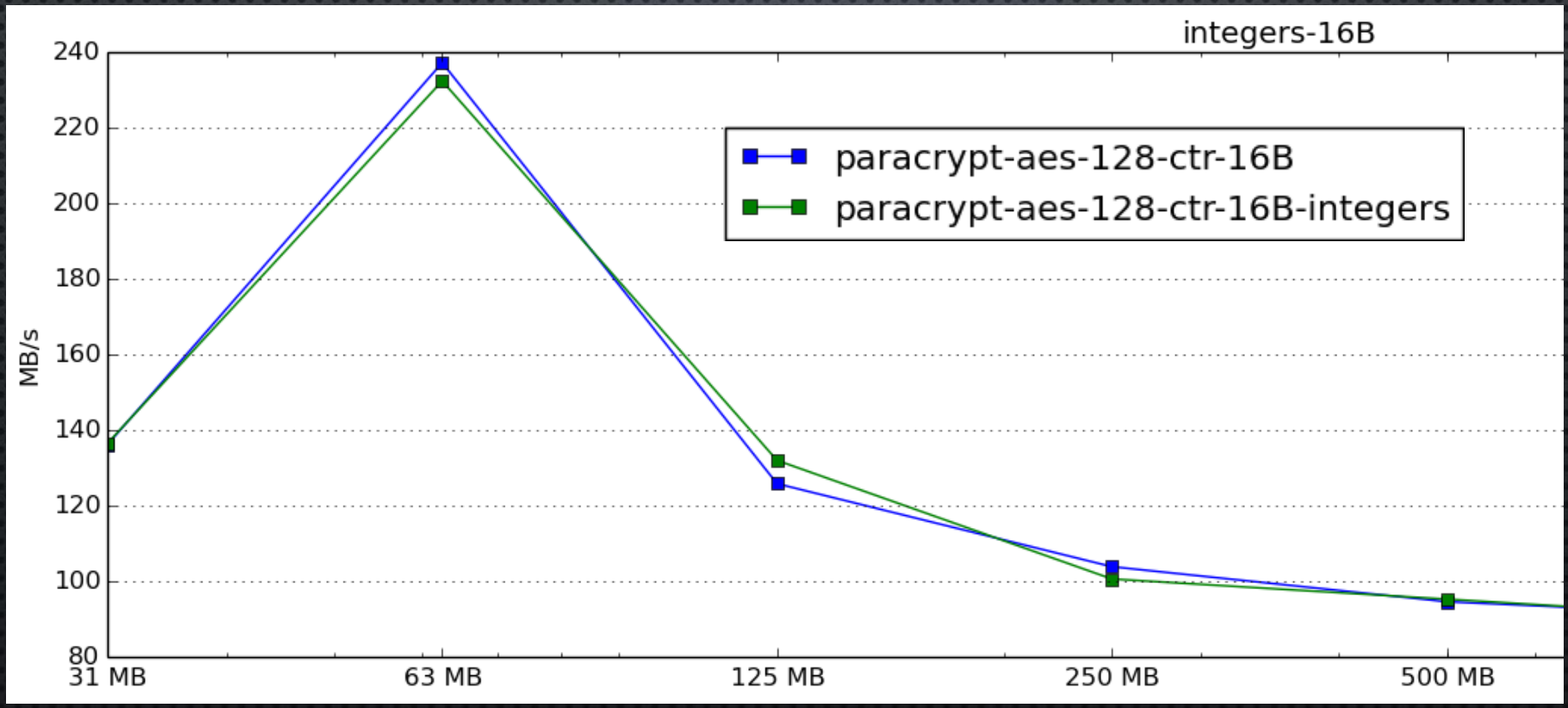
Operaciones con enteros

- No se han percibido diferencias significantes de rendimiento



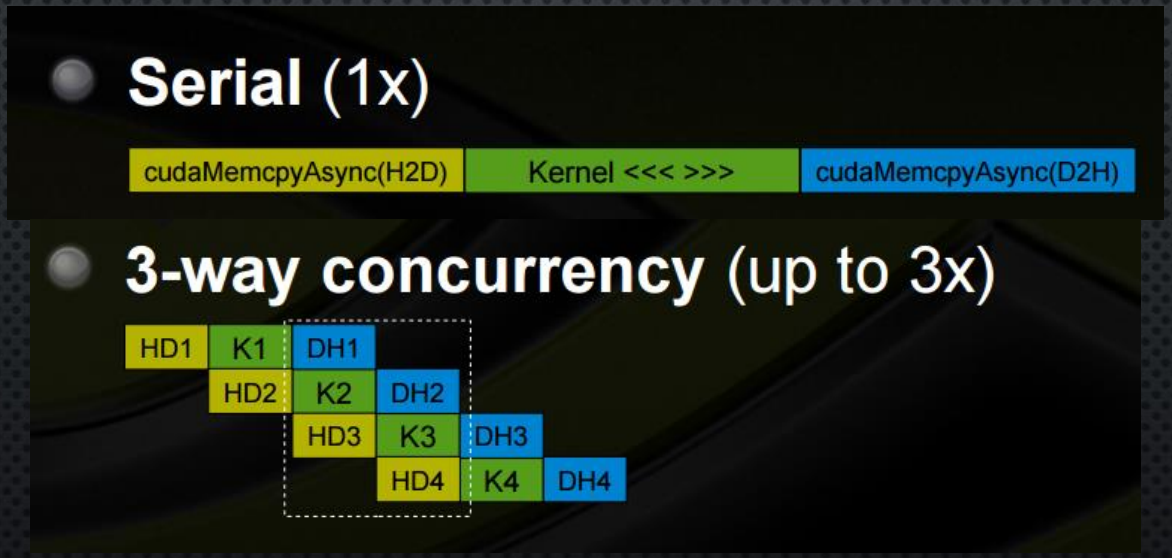
4.4 ACCESO BYTES ESTADO

Operaciones con enteros



4.5 TRANSFERENCIAS HOST-GPU

- Se ha tenido en cuenta solapamiento de computo y transferencias de datos a la GPU lanzando varios *kernels* (funciones que se ejecutan en la GPU) asíncronos en varios *streams (colas de operaciones independientes)*

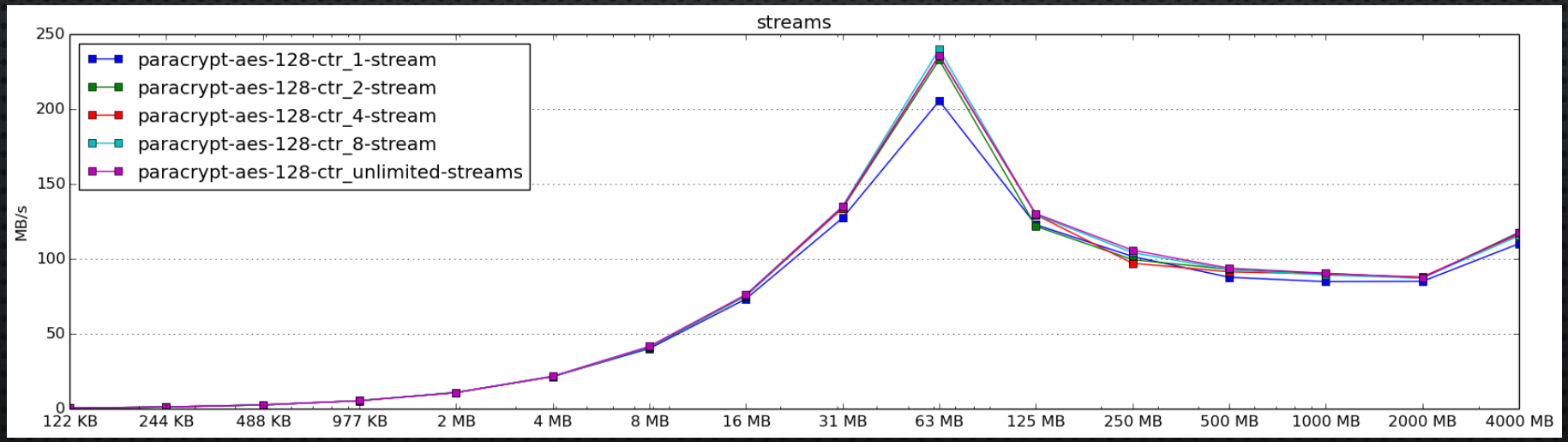


4.5 TRANSFERENCIAS HOST-GPU

- Solapamiento de computo y transferencias mejora el rendimiento hasta un 7.32%

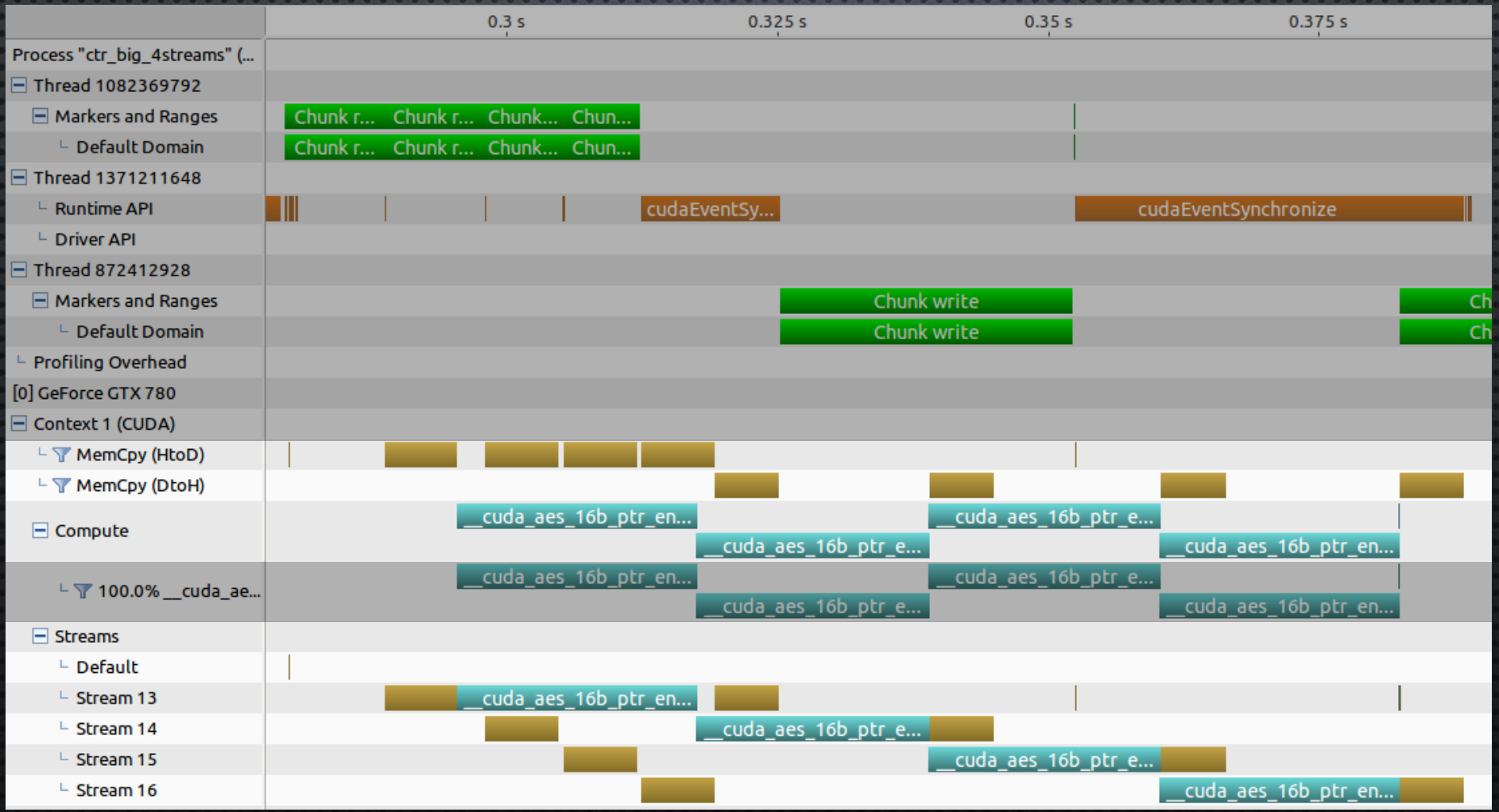
Media MB/s ficheros entre 16MB y 4GB

			∇ 1-stream
1	Stream	111.18	0 %
2	streams	117.10	5.32 %
4	streams	117.79	5.94 %
8	streams	119.07	7.09 %
16	streams	119.33	7.32 %



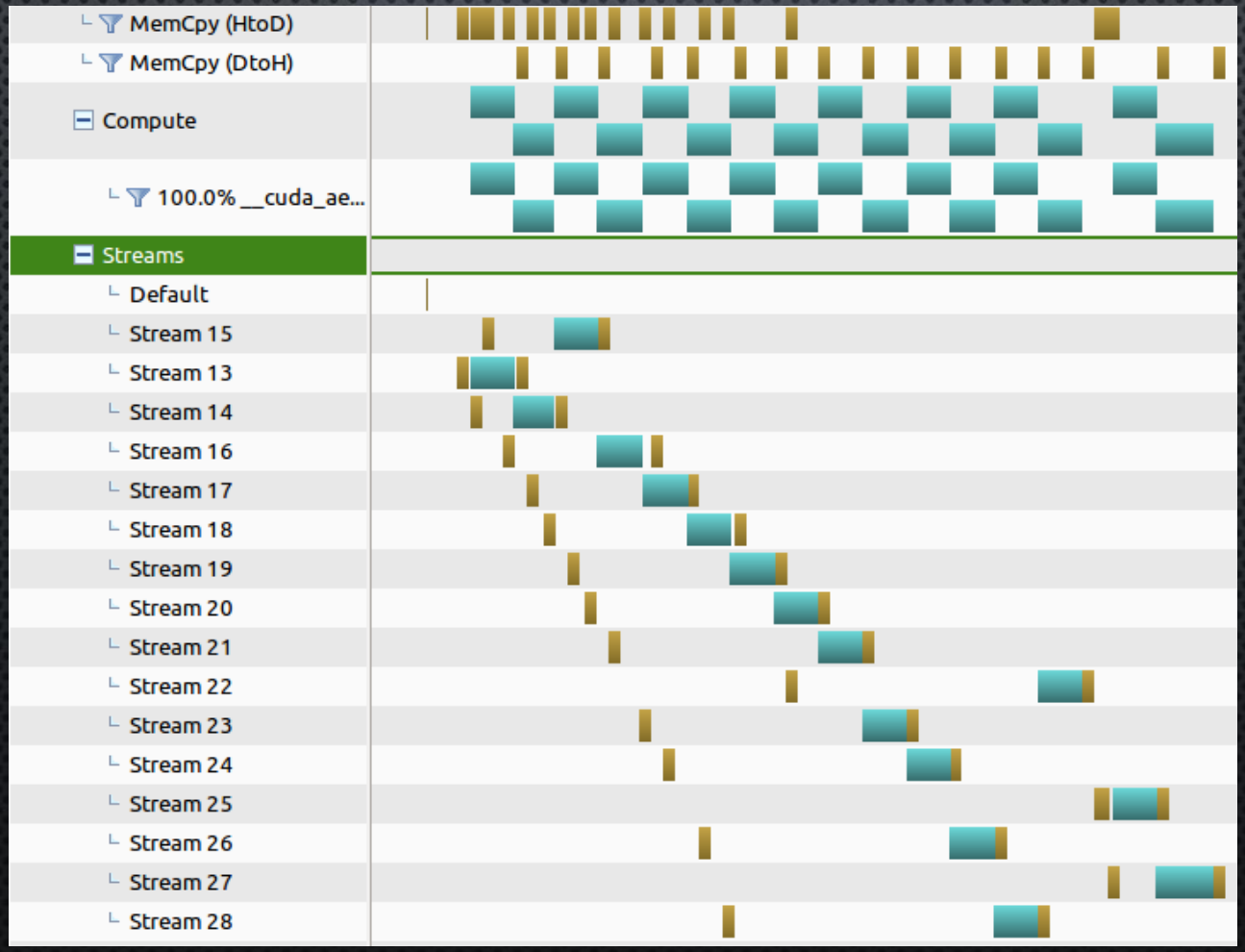
4.5 TRANSFERENCIAS HOST-GPU

Solapamiento con 4 streams



4.5 TRANSFERENCIAS HOST-GPU

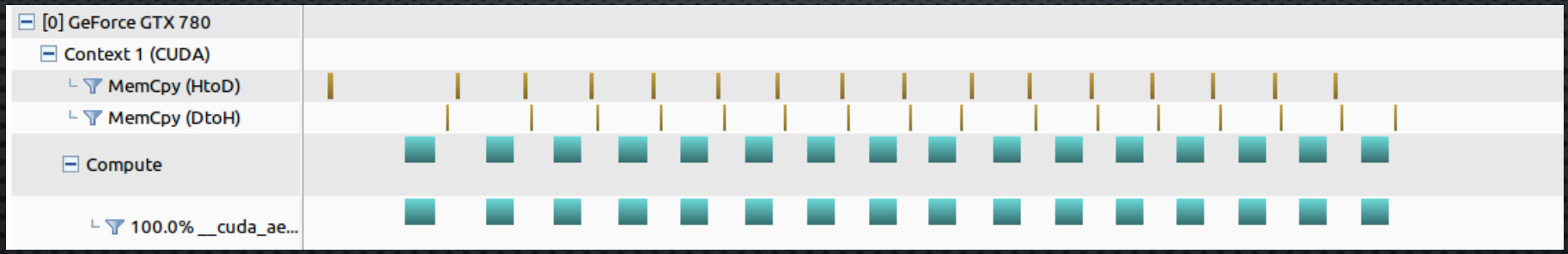
Solapamiento con 16 streams



4.5 TRANSFERENCIAS HOST-GPU

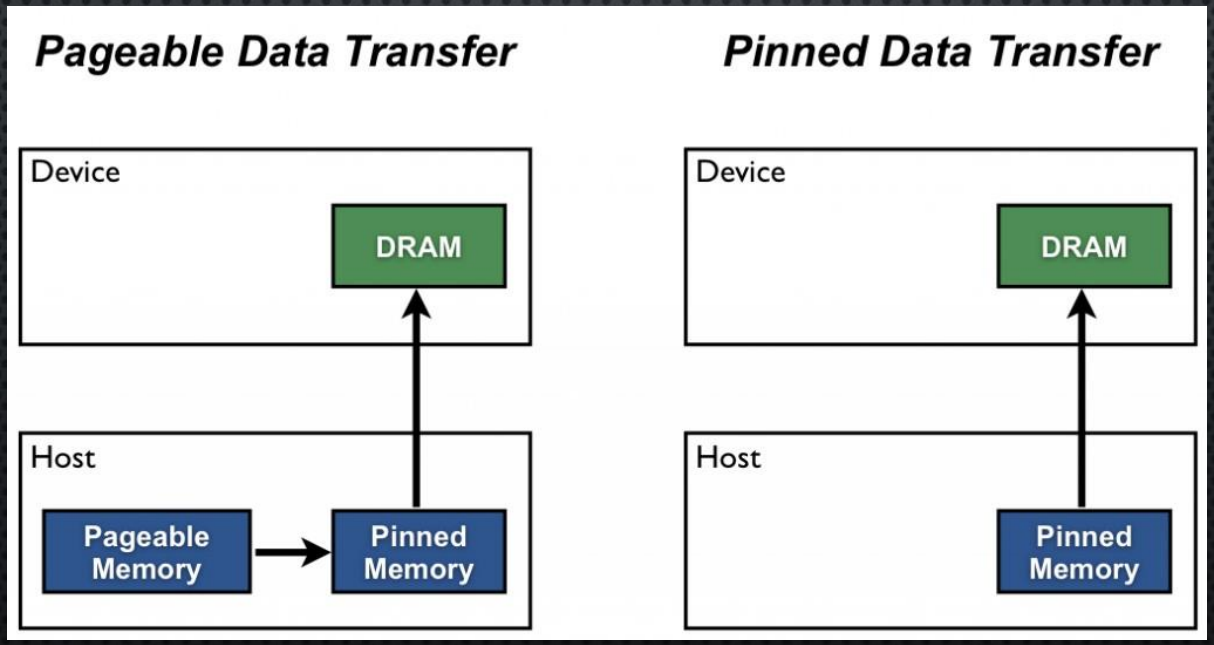
- Con ficheros pequeños el solapamiento se rompe
- Otra razón por la que no se le saca rendimiento a la GPU con pocos datos

Ej. Con fichero de 1.6 KB



4.5 TRANSFERENCIAS HOST-GPU

- El buffer de lectura del fichero se almacena en memoria no paginable permitiendo un mayor ancho de banda en las transferencias host-GPU



4.5 TRANSFERENCIAS HOST-GPU

- Un tamaño demasiado pequeño de buffer afecta negativamente al rendimiento al realizar demasiadas operaciones de copia host-GPU
- Un tamaño demasiado grande también afecta negativamente (coste de malloc sobrepasa beneficios)
- La implementación usa un buffer de 8MB por defecto

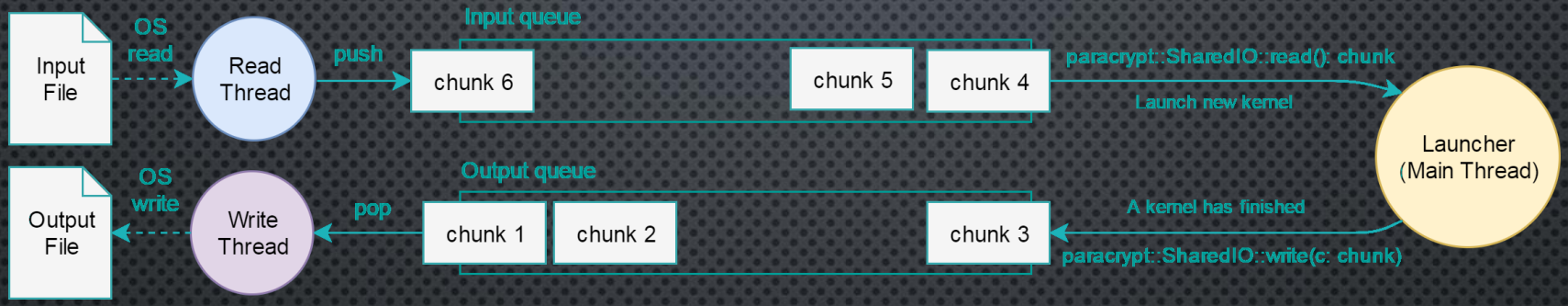
MB/s vs Tamaño fichero cifrado

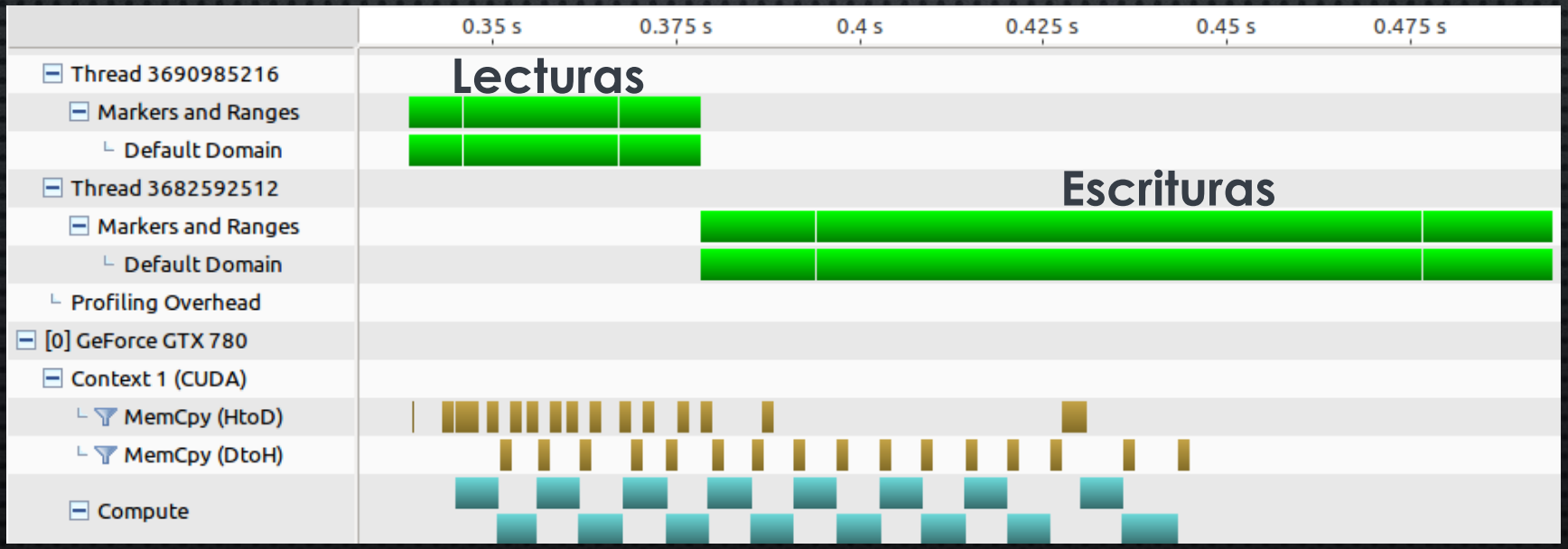
	4 GB	2 GB	1 GB	500 MB	250 MB	125 MB	63 MB	media
1MB	113.34	85.87	85.67	91.19	97.45	130.37	200.12	114.85
2MB	116.38	86.32	88.89	90.58	94.11	116.39	221.26	116.27
8MB	118.28	88.12	90.65	93.84	104.55	124.91	240.05	**122.91**
32MB	117.28	88.24	90.02	93.68	102.07	125.2	234.84	121.61
sin límite*	115.87	86.19	88.83	93.45	103.65	126.79	219.67	119.20

*max. ram disponible

4.5 TRANSFERENCIAS HOST-GPU

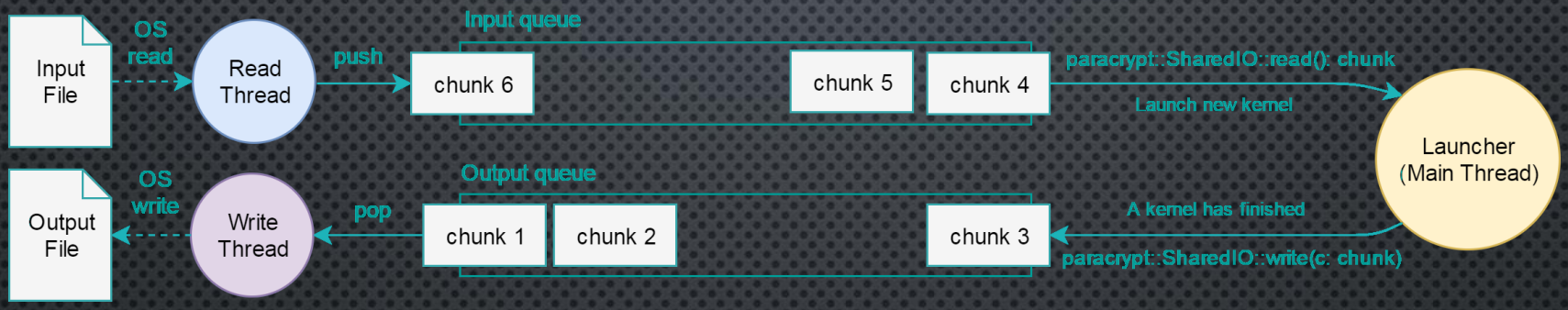
Transferencias host-GPU: Lecturas asíncronas del fichero de entrada





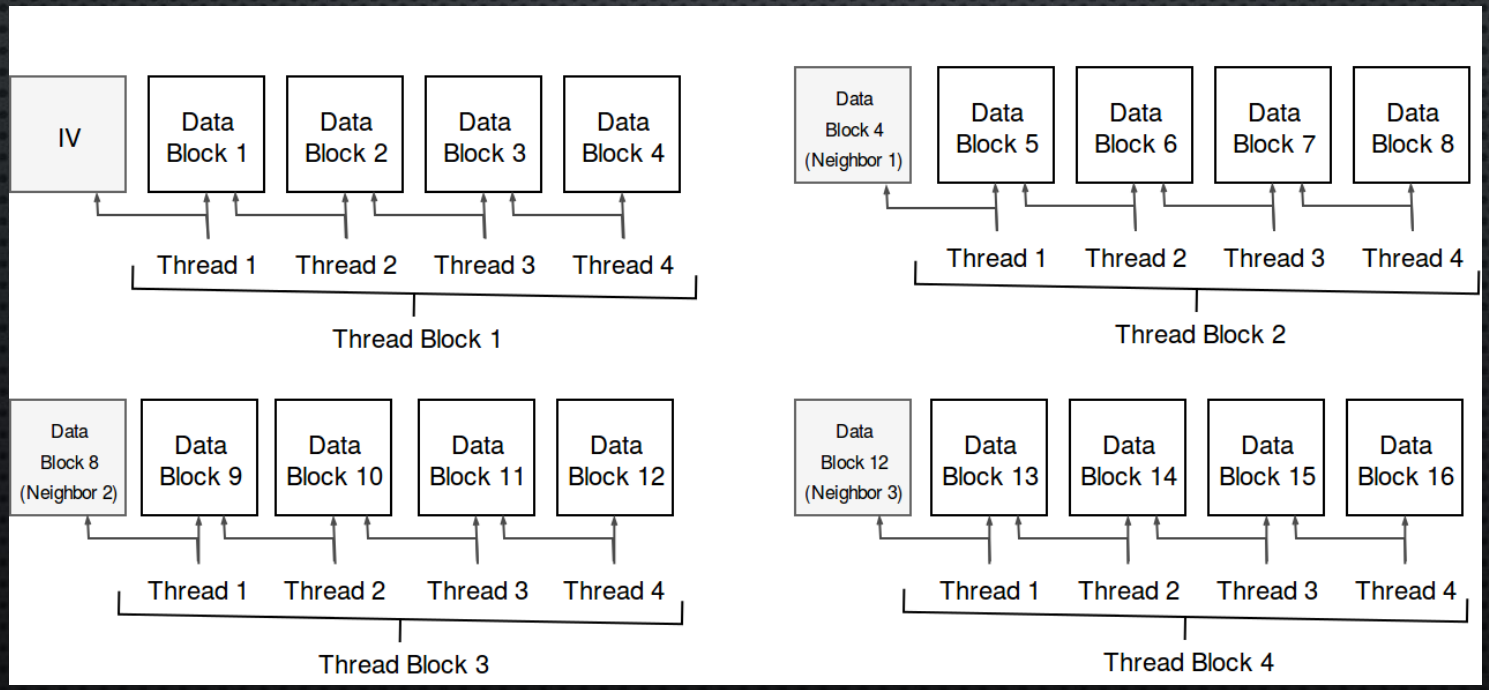
4.5 TRANSFERENCIAS HOST-GPU

Lecturas asíncronas del fichero de entrada



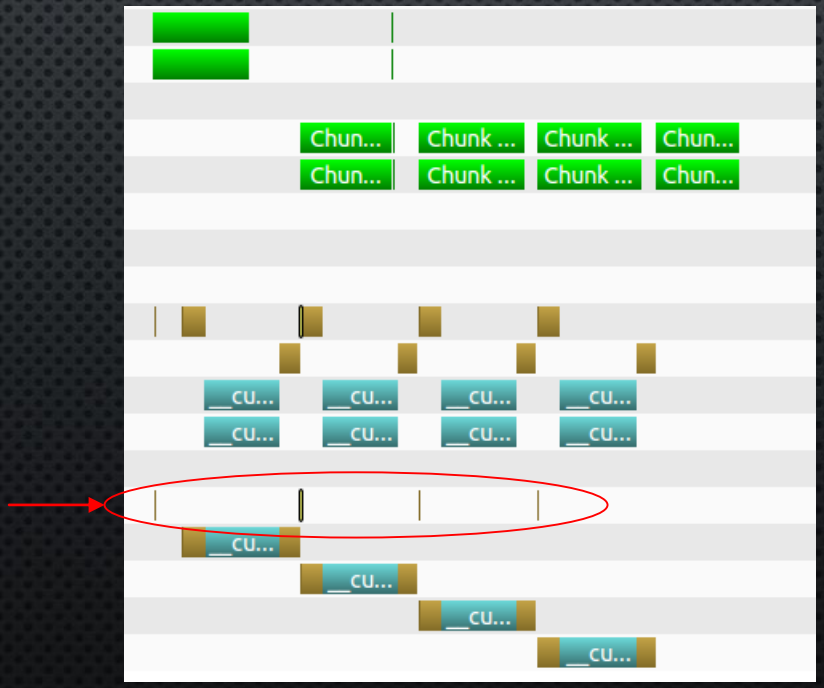
4.5 TRANSFERENCIAS HOST-GPU

- No hay forma ligera de sincronizar threads de pertenecientes a distintos bloques → condiciones carrera acceso bloques vecinos modos CBC y CFB
- Por ello, en el modo CBC y CFB se realizan copias adicionales de solo lectura (vecinos o *neighbors*) de algunos bloques



4.5 TRANSFERENCIAS HOST-GPU

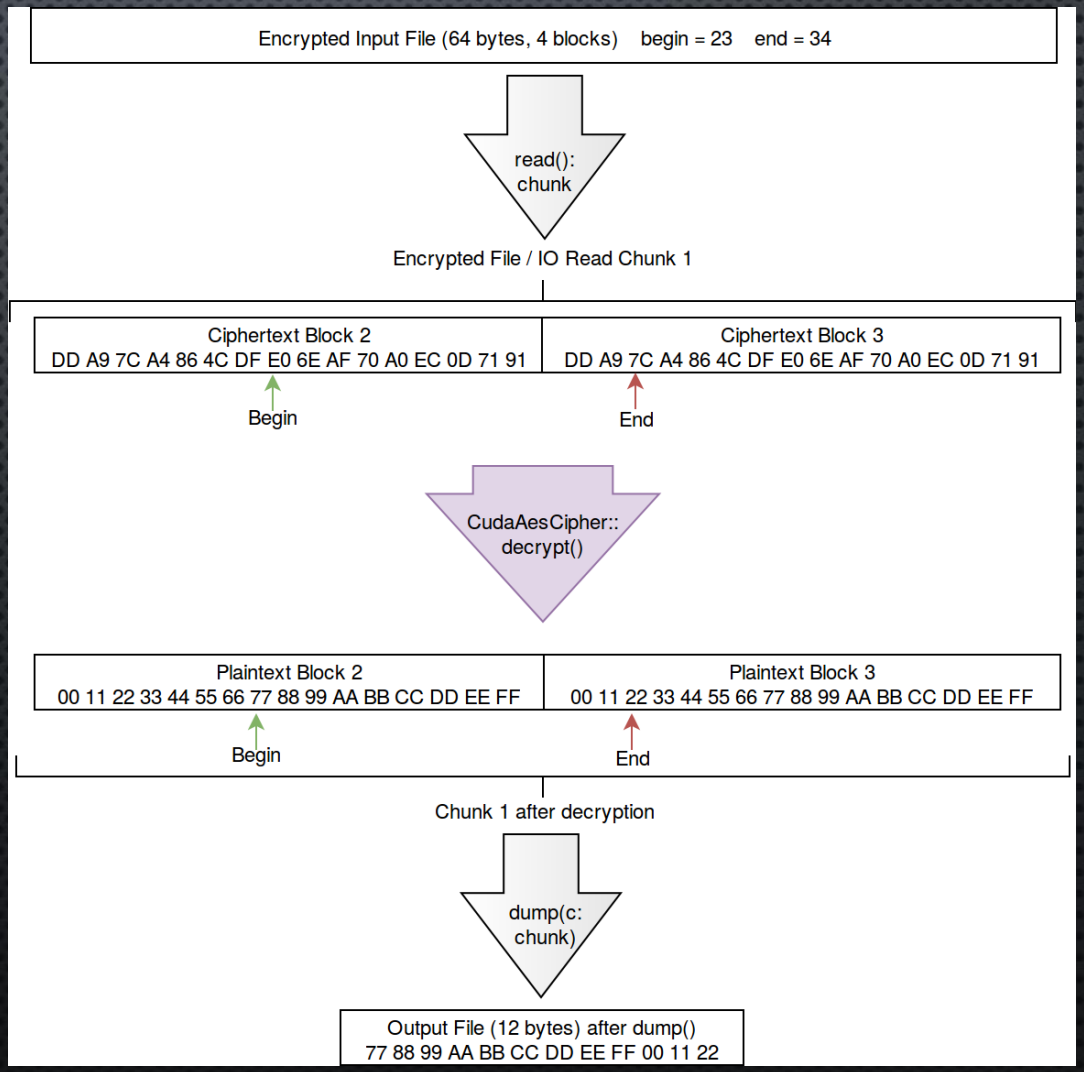
- Mejorable: El uso de una operación de transferencia separada para copiar vecinos del host a la GPU causa la ruptura de solapamiento computo y transferencias.
  → Generar copias de vecinos directamente en la GPU sobre memoria compartida



4.6 ACCESO ALEATORIO

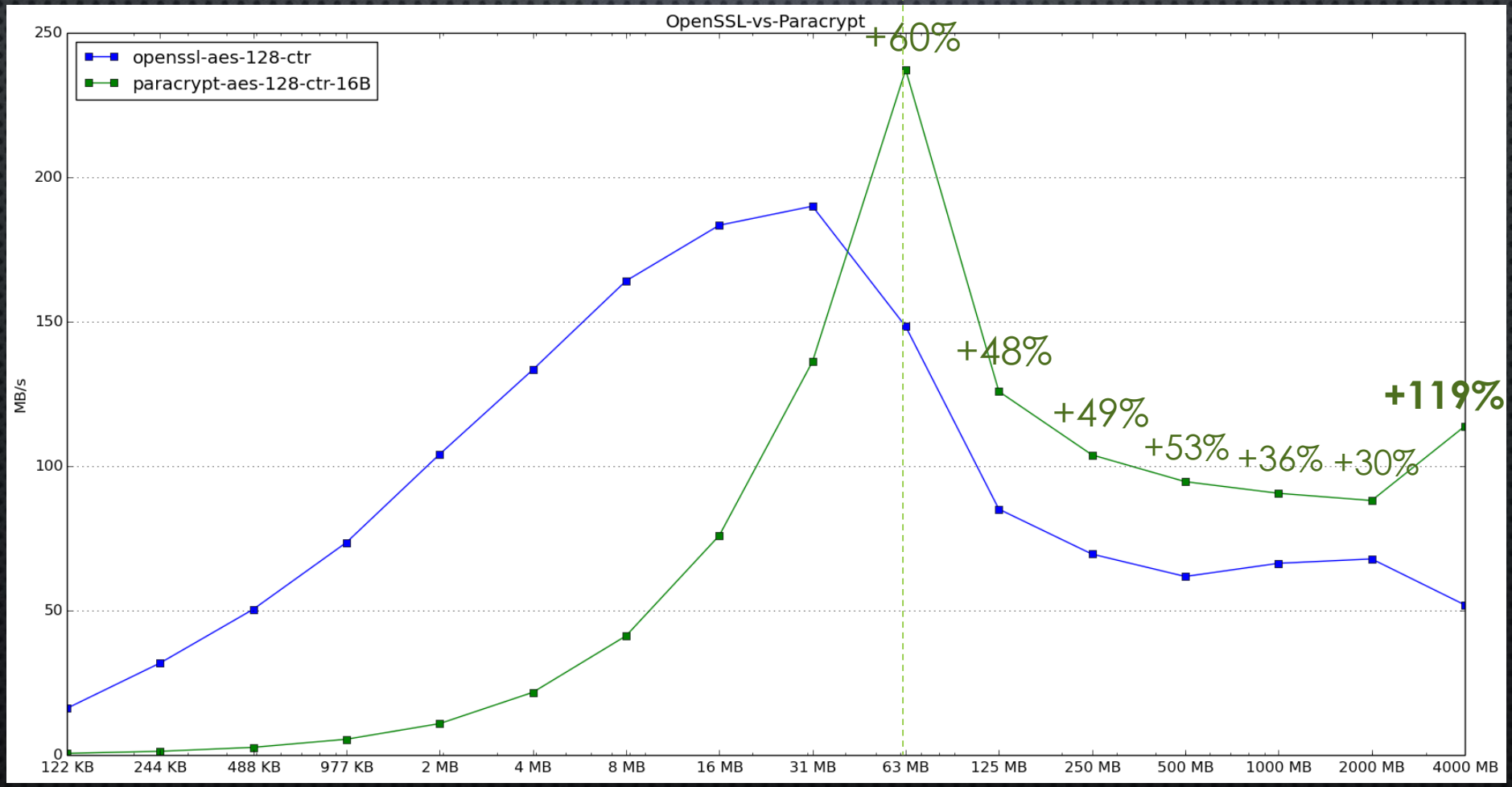
- Se ha implementado soporte para acceso aleatorio:

→ Obtener parte del mensaje en claro sin necesidad de descifrarlo entero



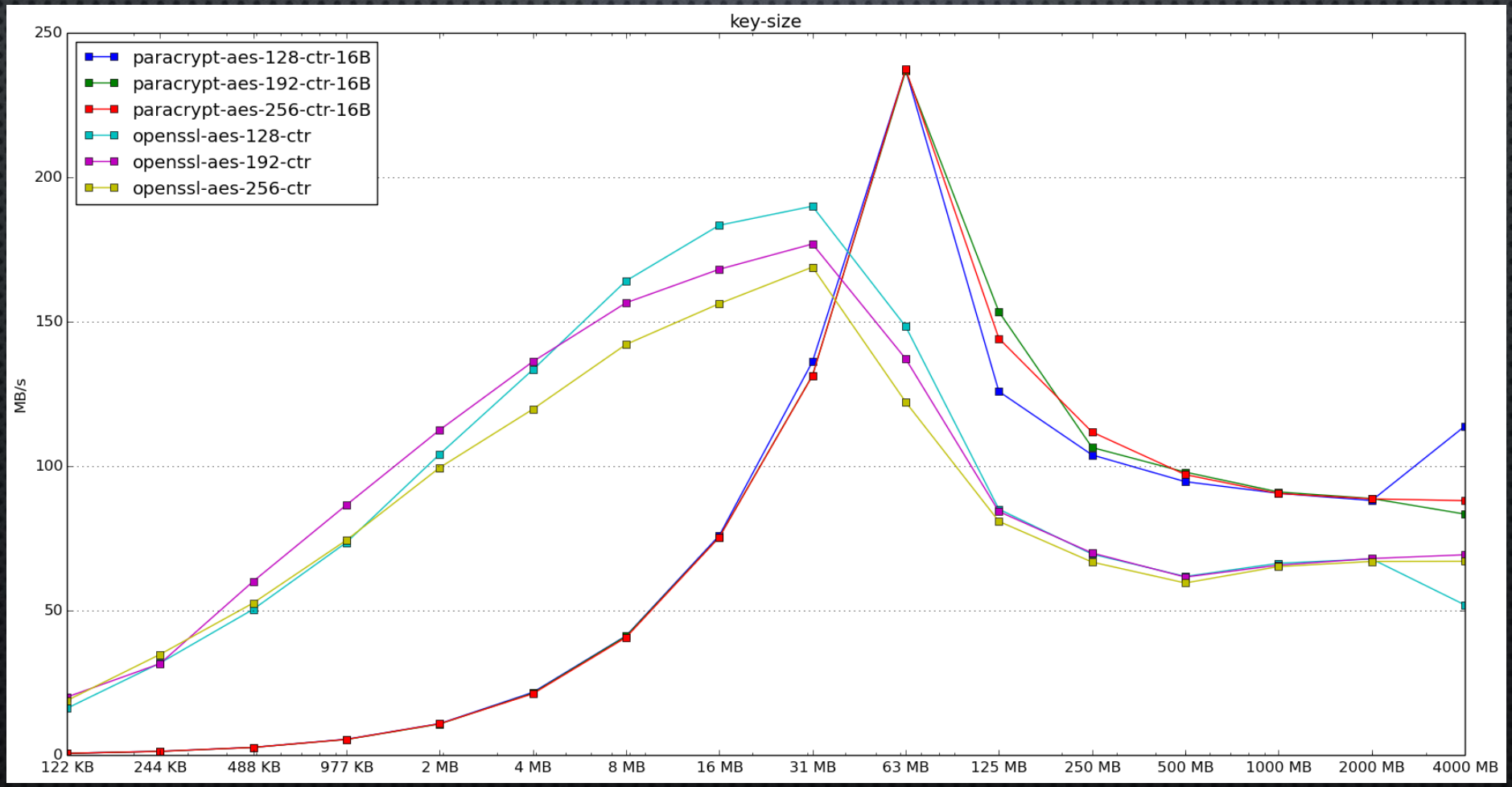
5. COMPARACION CON OPENSSL

Cifrado



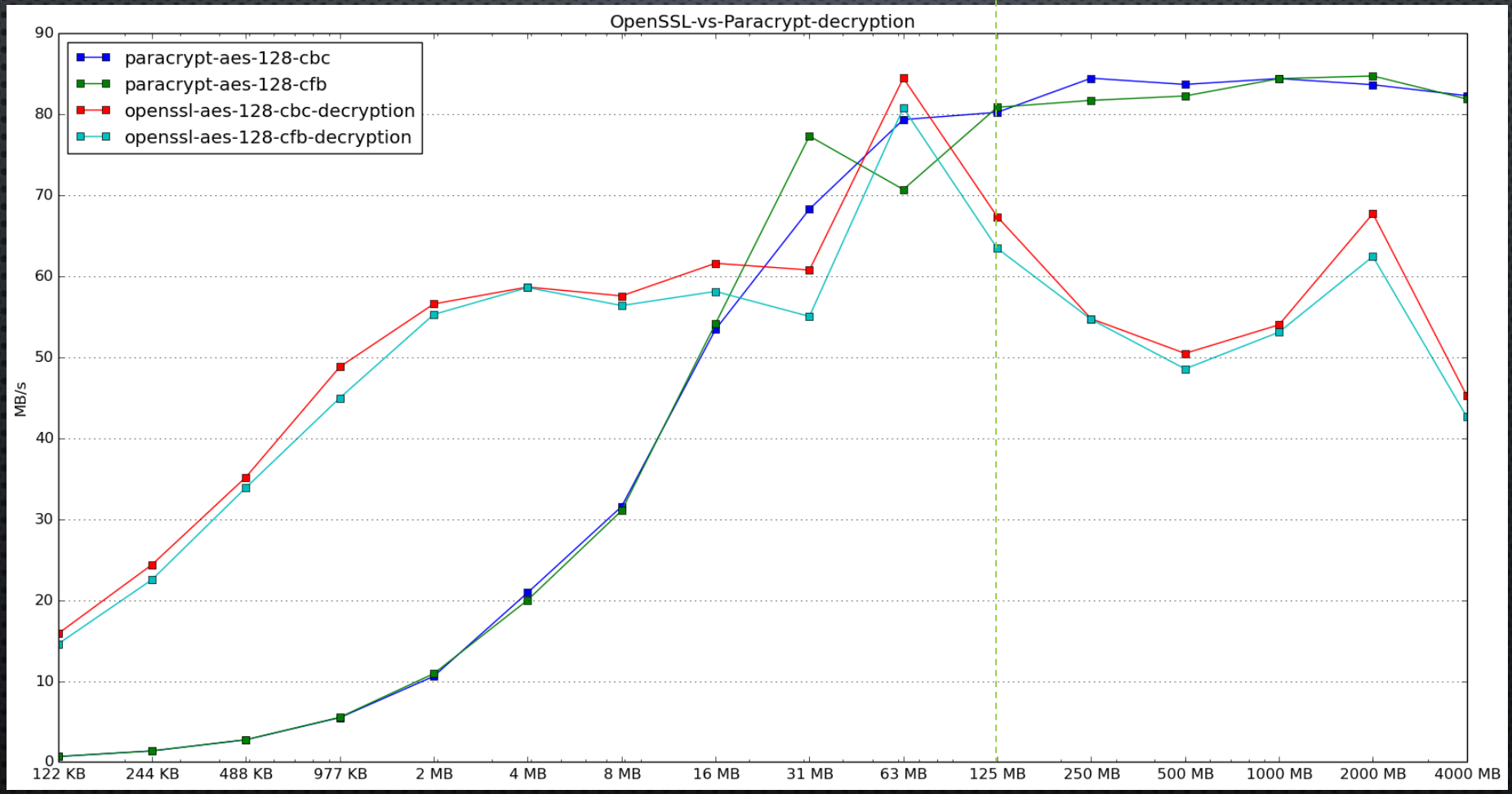
5. COMPARACION CON OPENSSL

Cifrado con distintos tamaños de clave



5. COMPARACION CON OPENSSL

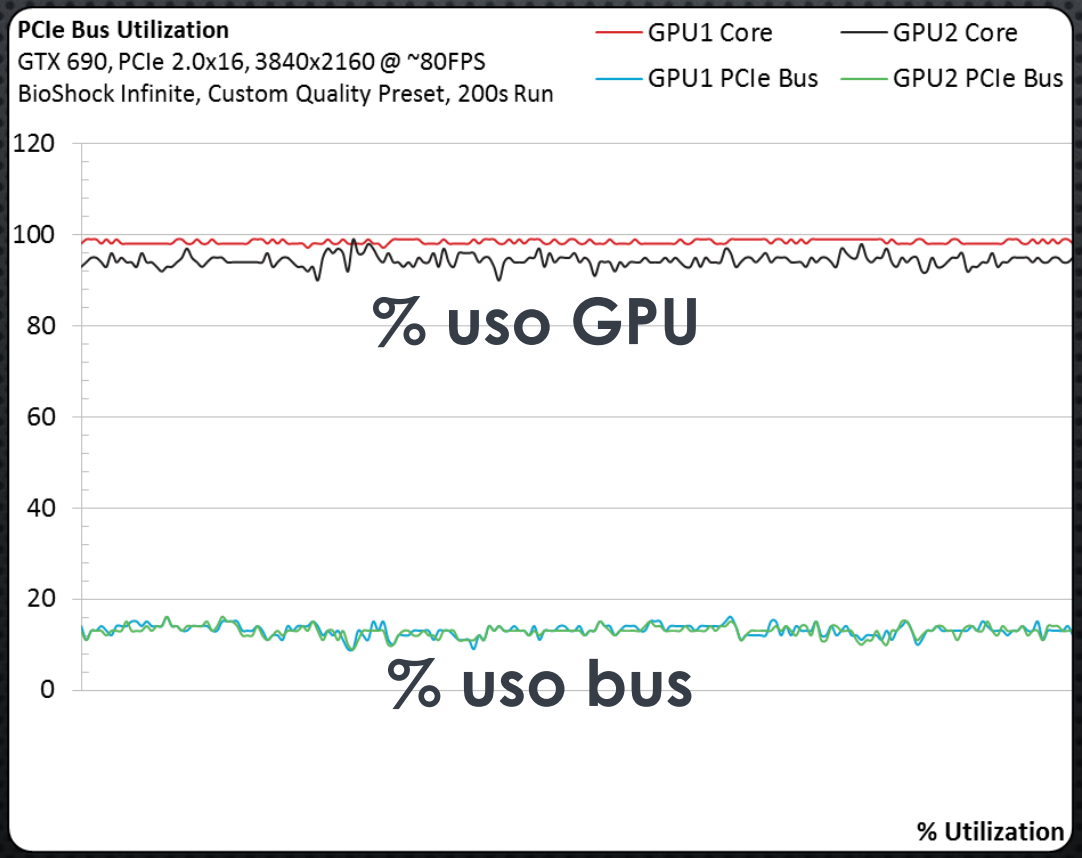
Descifrado



6. CONCLUSIONES

Ya hay implementada una versión funcional pero hace falta:

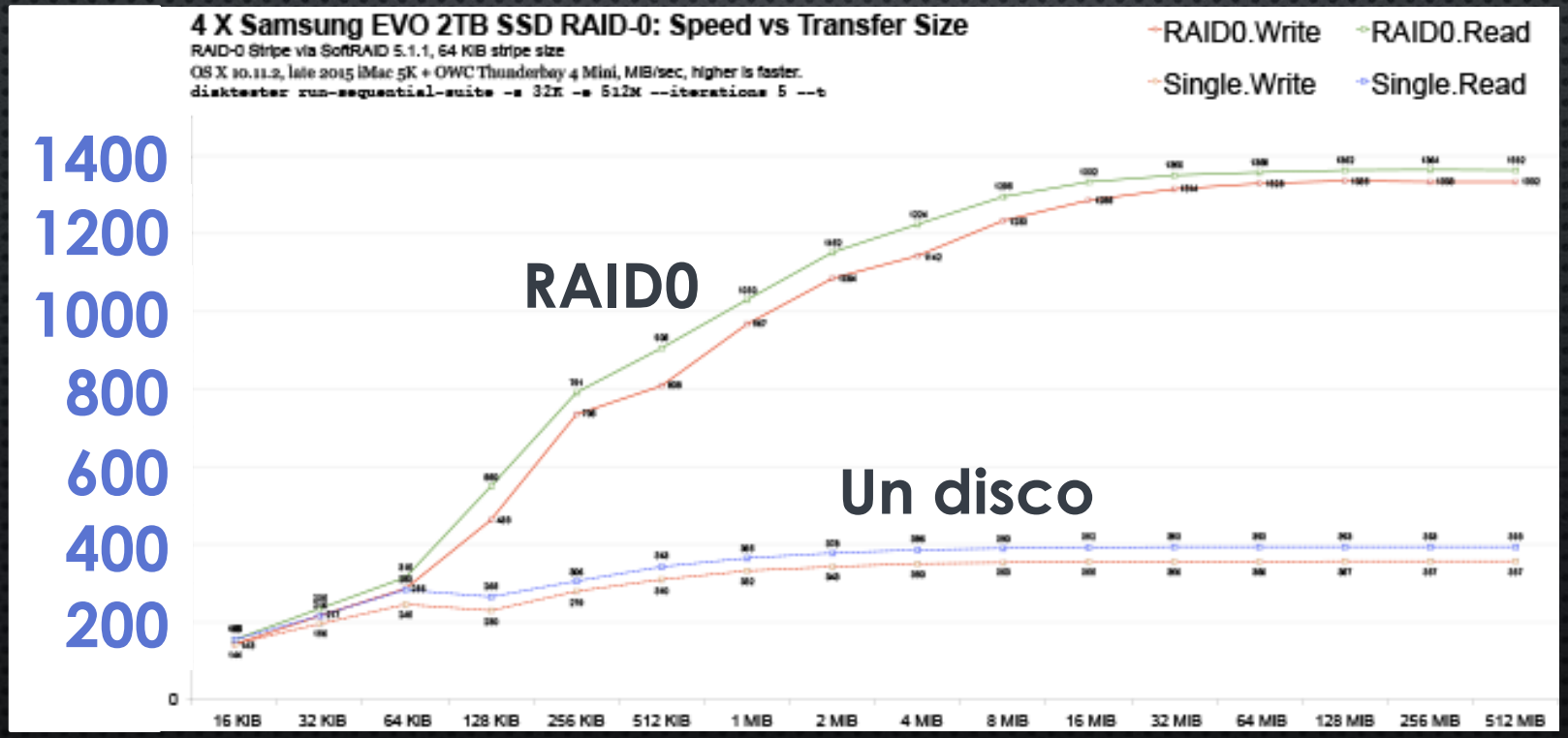
- Comprobar si estamos aprovechando todo el ancho de banda disponible por el bus o si hay cuello de botella en cómputo o disco



6. CONCLUSIONES

Ya hay implementada una versión funcional pero hace falta:

- Analizar si el disco es el cuello de botella (lo mas probable)
- Pruebas en sistemas con acceso secuencial a disco más rápido (por ejemplo en RAID)



6. CONCLUSIONES

Ya hay implementada una versión funcional pero hace falta:

- Es de esperar un mayor rendimiento al mejorar prestaciones del disco principal y poder cargar más datos en la GPU

Máquina de pruebas:

- Velocidad de lectura secuencial en disco: 208 MB/s
- Capacidad bus PCI Express x16 Gen2: 8 GB/s en cada dirección

¡Gran desaprovechamiento potencial de la GPU!

6. CONCLUSIONES

- 208 MB/s disco vs 8 GB/s bus
  ¡Desaprovechamiento potencial de la GPU!
- Hay que verificar que un mayor rendimiento es posible usando discos de mas prestaciones antes de descartar soluciones mas baratas como AES-NI para su uso en sistemas especializados

% beneficio esperado vs CPU:

+55% (+119% max.) cifrado
+49% (+87% max.) descifrado

% beneficio esperado AES-NI:

+38% cifrado
+37% descifrado

- La implementación se ha desarrollado partiendo de cero
  - Implementación OpenSSL como referencia
  - Solo se ha reutilizado el código de expansión de clave de la librería OpenSSL
  - Se ha diseñado/programado para funcionar con multiples GPUs aunque no se ha testeado aún

- El código desarrollado se puede compilar en Linux en una herramienta para la línea de comandos o como una librería compartida para su uso en C/C++

```
> paracrypt --help
GPU accelerated AES

Use: paracrypt -c cipher (-e|-d) -K hexadecimal_key [-iv input_vector] -in input_file -out output_file

Allowed options:
  -h [ --help ]                 produce help message
  --show arg                    show license warranty (w) or conditions (c)
  -q [ --quiet ]                disables logging engine and do not output any
                                messages
  -v [ --verbose ] arg          level of verbosity: warning (default), info,
                                debug, trace
  -c [ --cipher ] arg           selects one of the following ciphers:
                                aes-128-ecb aes-256-ecb aes-192-ecb aes-128-cbc
                                aes-192-cbc aes-256-cbc aes-128-cfb aes-192-cfb
                                aes-256-cfb aes-192-ctr aes-128-ctr aes-256-ctr
  -e [ --passwd ]               passwd input
```

- Se ha hecho uso de la biblioteca Boost para
  - Tests Unitarios
    - Pruebas cifrado/descifrado con vectores de prueba del FIPS 197 (estándar del NIST)
    - Cifrado de mensajes aleatorios y comprobaciones de que se obtiene el mismo mensaje tras descifrar
  - Distintos niveles de mensajes de log
  - *Parsing* de opciones de la utilidad de comandos
  - Funciones de sincronización: procesos ligeros y cerrojos

GPU ACCELERATED AES

IMPLEMENTACIÓN EN GPU DEL CIFRADOR AES

FIN

GRACIAS POR SU ATENCIÓN

AUTOR: JESÚS MARTÍN BERLANGA

TUTOR: JESÚS MARTÍNEZ MATEO

POLITÉCNICA