Estrategias para acelerar bases de datos mediante tarjetas gráficas

David Edgardo Castillo Rodríguez
ITAM, México
david.castillo@itam.mx

***Resumen*—En este ensayo se exponen los principales enfoques que se han propuesto e implementado para el aceleramiento de bases de datos al utilizar una tarjeta gráfica como coprocesador. En particular, se revisaron cuatro enfoques, los cuales buscan resolver o esconder el problema inherente al overhead generado por las transiciones de memoria entre el CPU y el GPU. El primer enfoque, *Base de datos en la memoria GPU*, esconde el overhead al cargar la base de datos o parte de esta en la memoria de la GPU, para hacer esto se sacrifica la posible escalabilidad de la implementación. El segundo enfoque, *Compresión de datos*, esconde el overhead al realizar transferencias de memoria de información comprimida, pero se introduce un nuevo overhead asociado a los algoritmos de compresión-descompresión. El tercer enfoque, *Arquitectura híbridas*, esconde el overhead al realizar transiciones de memoria de manera concurrente, el principal reto en este enfoque es cómo coordinar tanto al CPU como al GPU para que trabajen de manera cooperativa. El cuarto enfoque, *Arquitecturas integradas*, elimina el overhead al introducir una memoria compartida entre el CPU y el GPU, pero se agrega un nuevo overhead al considerar protocolos de consistencia de memoria.**

***Index Terms*—CPU-GPU, CUDA, OpenGL, Bases de datos, Compresión de datos, Arquitecturas híbridas, Arquitecturas acopladas.**

I. Introducción

Hoy en día hay muchas aplicaciones, ya sea de uso comercial o científico, que generan una gran cantidad de información diariamente, como Amazon, Facebook, Whastapp, el acelerador de partículas, por ejemplo. De acuerdo a las estadísticas dadas en [1], diariamente son enviados 500 millones de tweets, 294 billones de mails, y 4 petabytes de información es creada en facebook, por ejemplo. También, de acuerdo a [1], se estima que para el año 2025 se generen 463 exabytes de información a nivel global diariamente (un exabyte equivale a $1,000^6$ bytes). En muchas ocasiones, con los datos obtenidos se nutren modelos de aprendizaje de máquina ya sea para encontrar algún patrón interesante en los datos o para ayudar a la toma de decisiones de negocio. Para poder utilizar toda la información que pueden brindar los datos, es necesario administrarlos mediante algún sistema de bases de datos, donde el tiempo de procesamiento para alguna tarea en particular es generalmente proporcional a la cantidad de información que se desea procesar. Para aplicaciones que requieren dar una respuesta rápida, considerando una gran cantidad de información, se ha aprovechado el paralelismo que ofrecen los multiprocesadores, así como la arquitectura SIMD, por ejemplo en [2] se propone cómo aprovechar las arquitecturas multicore para el caso de aplicaciones que fueron diseñadas para trabajar con un único hilo, en [3] se explora cómo explotar las arquitecturas multicore para eficientizar algoritmos de productos matriciales, y en [4] se implementan redes convulocionales en arquitecturas SIMD. Otra manera de obtener paralelismo es mediante las tarjetas gráficas, ya que estas han sido diseñadas para ofrecer grandes niveles de paralelismo de manera natural, lo cual ha llevado a utilizarlas para acelerar aplicaciones computacionalmente intensivas. Con respecto a las bases de datos, al tratarse de un gran volumen de datos, las tarjetas gráficas se están utilizando como coprocesadores para acelerar varias de las operaciones diarias que estas realizan, como en [5], [6] y [7] (en los cuales se ahonda más adelante), por ejemplo. Dicho lo anterior, en este ensayo se revisan los enfoques que se han seguido para poder incorporar a las tarjetas gráficas como coprocesadores. En la sección 2 se describe cómo surgieron las tarjetas gráficas. En la sección 3 se explica qué es CUDA. En la sección 4 se inlcuye un análisis de las estrategias para acelerar bases de datos mediante tarjetas gráficas. En la sección 5 se comentan las conclusiones obtenidas. Por último, en la sección 6, se expone el trabajo futuro.

II. Contexto histórico

Las unidades de procesamiento gráfico conocidas por sus siglas en inglés como GPUs (graphics processing unit), tienen sus origenes desde finales de los años 80, cuando los sistemas operativos con interfaz gráfica, como Microsoft Windows, dieron cabida a la creación de los aceleradores gráficos 2D (antecesores de los GPUs), estos aceleradores tenían un conjunto especial de instrucciones a nivel hardware el cual les permitia realizar tareas como graficar líneas y superficies en dos y tres dimensiones. Por la misma fecha, la empresa fabricante de hardware y software Silicon Graphics, popularizó el uso de gráficos en tercera dimensión en una gran variedad de aplicaciones comerciales, y en 1992 liberó al público la interface de programación de su hardware gráfico mediante la librería OpenGL, la cual pretendia ser un estándar para hacer aplicaciones en tercera dimensión.

Dado el éxito que comenzaban a tener los aceladores gráficos, estos siguieron desarrollándose hasta convertirse en lo que hoy en día conocemos como GPUs. En un principio (al rededor del año 2000), estos fueron diseñados para producir el color para cada pixel de un monitor de video mediante una unidad aritmética programable conocida como *pixel shader*. Estas podían recibir como entrada practicamente cualquier dato numérico, lo que llevo a que las GPUs no fueran utilizadas unicamente para producir colores, sino que también para ejecutar programas.

En un inicio era complicado hacer cómputo de propósito general en las GPUs, porque la única manera de comunicarse

con estas era através de unas cuantas interfaces como la de OpenGL, las cuales estaban restringidas a la manera en que un programa pudiera escribir en memoria, o que no hubiera una forma efectiva de encontrar errores en los programas, por ejemplo. No fue hasta el año 2006 cuando la empresa *NVIDIA* lanzó al mercado una tarjeta gráfica basada en su también arquitectura *CUDA*, la cual incluyó varios componentes diseñados para hacer cómputo en la tarjeta, y a su vez, esta arquitectura resolvió varias limitantes de las interfaces de programación gráfica, para así poder implementar programas correctos de propósito general dentro de sus GPUs, es posible leer a más detalle acerca cómo surgieron las GPUs, así como de CUDA en [8] .

III. Aceleramiento con CUDA

Hoy en día muchas aplicaciones han sido implementadas o aceleradas en CUDA debido a que su arquitectura SIMT (simple input multiple thread) permite alcanzar mejor rendimiento mediante la ejecución de miles de hilos en paralelo. Para poder alcanzar este nivel de paralelismo, la arquitectura de CUDA requiere que el CPU (conocido como Host) le indique al GPU (conocido como device) qué kernel ejecutar, así como que transfiera a la memoria global del GPU los datos a ser ejecutados a través de un bus conocido como CPI. El kernel es dado por el usuario, y este consiste en código definido para ser ejecutado en paralelo dentro de los hilos del GPU.

En la literatura es posible encontrar reportes sobre aplicaciones aceleradas mediante el uso de una GPU, muchas de estas aplicaciones reportan mejoras en el tiempo de ejecución en factores que van de $10\times$ hasta por $100\times$, pero como se menciona en [9] y [10] estas comparaciones podrían estar fuertemente sesgadas ya que no incorporan otros factores que pudieran influir en el resultado, como el tiempo de transferencia de memoria u optimizaciones a nivel software. En [9] se propone que para medir el aceleramiento de manera justa, es necesario que el código que se ejecute tanto en el CPU como en el GPU sean lo más óptimo posible, para así aprovechar la arquitectura SIMD de los multiprocesadores o SIMT de las GPUs con una mayor eficiencia. Mediante estas optimizaciones de código, aplicaciones que reportaban aceleramientos en la GPU en factores de hasta $100\times$, realmente tienen un aceleramiento promedio en un factor de $2.5\times$. Por otro lado, en [10] se menciona que no solamente es importante considerar hacer comparaciones de manera optimizada, sino que además, es necesario analizar el tiempo de transferencia de memoria, ya que cuando este es incluido en la medición del tiempo de ejecución de un kernel, el tiempo de ejecución se incrementa en un factor de entre $2\times$ a $50\times$ en comparación a no considerar el tiempo que tardan las transferencias de memoria. Por ello en [10] se sugiere que para reportar de manera adecuada el aceleramiento de alguna aplicación ejecutada en la GPU, es necesario reportar la latencia resultante de las transiciones de memoria, o justificar por qué esta no es relevante para la aplicación.

IV. Metodologías de aceleramiento

En la literatura se pueden encontrar diversas metodologías para acelerar algún tipo de base de datos utilizando como coprocesador una GPU, en particular, cada una de las metodologías implementadas se puede estudiar a partir de la idea de cómo reducir o eliminar el overhead inherente a las transiciones de memoria entre el CPU y la GPU, porque este representa el principal cuello de botella, ya que como se menciona en [10], este puede llegar a representar hasta un $80\,\%$ del tiempo total de ejecución. En términos generales, estas metodologías se fundamentan en alguna de las siguientes estrategias:

A. Base de datos en la memoria de la GPU.
B. Compresión de datos.
C. Arquitecturas híbridas.
D. Arquitecturas acopladas.

IV-A. Base de datos en la memoria de la GPU

Una de las maneras mediante las cuales se ha buscado evadir el overhead ocasionado entre las transferencias de memoria es mediante la transferencia de la base de datos a la memoria RAM global de la GPU, por ejemplo, en [5] se propone acelerar las operaciones SELECT del lenguaje SQL de una base de datos relacional, para ello, desarrollaron una interface la cual recibe código en SQL, el cual es parseado a un conjunto de códigos de operación, para posteriormente ser ejecutado en la GPU, de esta manera evitan realizar operaciones tipo SELECT en la base de datos mediante la invocación explícita de las primitivas de CUDA. La metodología anterior ayuda a aprovechar el gran paralelismo de la GPU, y además, ayuda a que un usuario de bases de datos relacionales no necesite aprender la sintaxis de CUDA. Un punto débil de este enfoque es que para poder llevar a cabo las operaciones, la base de datos debe cargarse en la memoria RAM de la GPU, lo cual no es práctico para bases de datos que superen el tamaño de la memoria RAM de la GPU.

En [6] se diseñan e implementan los algoritmos *Hash Join* y *Sort-Merge Join* donde se reporta que los algoritmos propuestos mejoran por un factor de entre $2.0\times$-$14.6\times$ y $4\times - 4.9\times$ a otras implementaciones en GPU de estos algoritmos, mientras que un factor de $10\times$ y $5.5\times$ para implementaciones de estos algoritmos implementados con CPU. Otro aspecto importante, es que estos algoritmos también han sido diseñados para el caso en que solo es posible mantener una base de datos en la memoria de la GPU, y la otra en el disco duro, obteniendo así más escabilidad al precio de incrementar la latencia, donde esta última puede esconderse mediante la concurrencia entre la ejecución del kernel y la transferencia de memoria. Para este caso, la base que no cabe en el GPU es particionada, y ejecutada de manera parcial en la GPU, cada que un resultado parcial obtenido por la GPU es devuelto al Host, este último envia concurrentemente a la GPU otra partición de la base para ser ejecutada. Para este último escenario, el factor de mejora para ambos algoritmos es de $3.5\times$ aproximadamente. A comparación de [5], se incorpora un caso en específico donde

una de las bases no cabe en la memoria principal de la GPU, lo cual podría no ser el caso para muchas bases de datos que se requieren procesar en el día a día.

En la misma línea que [6], en [7] se implementa el algoritmo *Hash Join* el cual se adapta tanto para el caso en que ambas bases de datos caben completamente en la memoria de la GPU, como para el caso en que ninguna cabe, obteniendo así un algoritmo completamente escalable. Para el caso en que ninguna base cabe en la memoria de la GPU, se utiliza un algoritmo de hash basado en coparticiones, donde el único requisito es que las coparticiones quepan en la memoria global de la GPU. Al igual que en [6] se esconde la latencia de la transferencia de memoria mediante la ejecución concurrente del kernel. Por último los autores comparan su metodología contra la base de datos híbrida DBMS-X, reportando un factor de mejoramiento de entre $1.5\times$ y $2\times$ para el caso en que las bases de datos caben completamente en la memoria global de la GPU, y un factor de $10\times$ cuando ninguna base cabe completamente en la memoria de la GPU. En [7] se propone un modelo capaz de procesar operaciones de tipo join para una base de datos de cualquier tamaño, tal vez sea interesante que esta pueda incorporar algún otro tipo de operaciones, como las [5] y [6], e inclusive que fuese adaptada para tener una aplicación capaz de parsear el lenguaje sql y aplicarlo sobre las primitivas de CUDA como se propone en [5]. También, para el caso en que alguna base de datos no quepa completamente en la memoria de la GPU tanto [6] como [7] se adaptan a una arquitectura híbrida, es decir, donde interactua tanto el CPU como el GPU.

IV-B. Compresión de datos

En [5], [6] y [7] se intenta evadir (o esconder) el overhed ocasionado por las transferencias de memoria al cargar las bases de datos (o parte de estas) en la memoria principal de la GPU, sin encambio, en [11] se es consciente de que las transferencias de memoria son necesarias. Y, por ello, se busca compensar este costo con el alto grado de paralelismo con el que se puede manipular una gran cantidad de información comprimida dentro de la GPU. En particular, en [11] se implementan nueve esquemas de compresión aplicados a bases de datos columnares, donde a la GPU se le envian arreglos de infromación, a estos se les aplica un algoritmo de compresión dentro de la GPU, posteriormente se aplica alguna operación, como de *Map*, *Scatter* o *Prefix Sum*, por ejemplo, y posteriormente el arreglo resultante es regresado al Host de manera descomprimida.

En [12] se menciona que muchas de las técnicas de compresión se enfocan en el intercambio entre la eficiencia de compresión y el tiempo que tarda el algoritmo, pero dejan fuera el tiempo de descompresión, lo cual podría ser más importante para grandes cargas de trabajo, por ello implementan *Gompresso*, la cual explota el paralelismo de la GPU para obtener mayores velocidades de descompresión. De manera resumida, *Gompreso* coordina eficientemente a los hilos que se encargan de descomprimir de manera concurrente al mismo bloque de información, de esta manera los autores reportan una mejora por un factor de $2\times$, en comparación con un CPU multicore con las librerías de descompresión más actuales, también se reporta un ahorro de energía de hasta un $17\,\%$ en el proceso. En particular, en [12] se estudia una variante del formato de compresión *DEFLATE*, la cual es usada en diversos formatos de archivos, como *.gzip*, *.zip* y *.png*, por ejemplo. Algo que tal vez no se tiene en consideración en [12], es que una vez que se ha logrado la descompresión, si esta se quiere manipular de manera eficiente dentro de la memoria de la GPU, estará limitada a la memoria de esta última, por lo que implementar un método eficiente de descompresión podría no ser suficientemente eficiente si no se toma en cuenta qué tipo de operaciones realizadas a los datos podrían tomar ventaja de esto. Dicho lo anterior, se podrían combinar las ideas de [11] y [12] para implementar una aplicación que sea eficiente tanto para comprimir, descomprimir y manipular la información.

En la misma línea que en [12], en [13] se presenta la librería *Giddy*, la cual continua hoy en día en desarrollo, esta es una librería de código abierto para la GPU, donde están implementados los algoritmos más usados de descompresión. Aunque esta librería continua en desarrollo, se basa en la idea de que una manera de disfrazar el overhead generado por las transiciones de memoria, es enviando de manera comprimida la información por el bus PCI, para posteriormente procesarla o descomprimirla dentro de la GPU. Al igual que en [12], no se muestra una aplicación donde esta información puede ser utilizada posteriormente, por lo que sería interesante incorporar algunas aplicaciones u operaciones como las implementadas en [11] para tener un mejor punto de comparación de cuando esta librería podría ser más eficiente que alguna contraparte implementada en un CPU.

IV-C. Arquitecturas híbridas

Como se menciona en [14], muchos enfoques de aceleramiento consideran que la GPU es un sistema ajeno al CPU, y como tal, no hay una cooperación eficiente entre ambos dispositivos, por ejemplo en [5] se busca procesar toda la base de datos en la memoria de la GPU dejando fuera al CPU, o por ejemplo en [13] se implementan librerías de descompresión adaptadas unicamente al GPU. Lo anterior motiva a que en [14] se proponga una estrategia donde tanto el CPU como la GPU cooperan para el procesamiento óptimo de consultas aproximadas de bases de datos relacionales. Para lograr la cooperación entre ambos dispositivos, en [14] se propone el paradigma *Approximate & Refine*, el cual consiste en particionar verticalmente los valores de una columna a una granuladidad de bits individuales, donde la parte más significativa de la partición es mandada a la memoria de la GPU, mientras que la menos significativa a la memoria principal, de esta manera la parte con más bits es una aproximación del verdadero valor. También se proponen los respectivos algoritmos para manipular la información que ha sido transformada mediante este enfoque, en particular se proponen los algoritmos de *Approximation* y *Refinement*, donde los primeros dan una aproximación rápida a la consulta deseada, mientras que los de refinamiento reconstruyen los verdaderos valores que dio

alguna operación de *Approximation* (concatenan los bits que se encuentran tanto en memoria principal como en la GPU), para después descartar a aquellos valores que podrían no cumplir con la condición de algún operando, y de esta manera se obtiene un resultado más acertado. Los autores utilizaron MonetDB para probar su implementación, donde reportan un aceleramiento de un factor de hasta $8\times$ en comparación de simplemente utilizar MonetDB sin considerar el paradigma *Approximate & Refine*.

Al igual que [11]-[13], este enfoque puede entenderse como una compresión de la información, donde esta es procesada en la GPU y posteriormente regresada a memoria principal para hacer algun refinamiento, y de esta manera aprovechar tanto al CPU como al GPU. Otro punto a considerar es que tal vez sea complicado medir la pérdida de información inherente a cada aproximación y refinamiento, lo cual puede ser muy sensible para cierto tipo de aplicaciones, como las bancarias, por ejemplo. Por lo que puede que este enfoque sea útil únicamente para ciertos escenarios.

Por otro lado, en [15], se propone utilizar a la GPU para acelerar las operaciones de bases de datos de tipo llave-valor en memoria principal (*In-memory key-value store, IMKV*), para ello se implementa Mega-KV, el cual es un sistema donde coopera tanto el CPU como el GPU para procesar consultas de tipo búsqueda, insertar y eliminar, en este tipo de base de datos. Al igual que en [11]-[12], se explota la capacidad de guardar información en la memoria principal de la GPU, pero con la diferencia de que en [15] únicamente se guarda una tabla hash como estructura de índices, donde los autores argumentan que una tabla hash de 3 GB es capaz de indexar 192 GB de información. Mega-KV divide el procesamiento de consultas en tres etapas, las cuales son: pre-procesamiento, procesamiento en GPU y post-procesamiento. En la primer etapa, el CPU es encargado de identificar y empaquetar un gran número de consultas según su tipo. En la segunda etapa, estas consultas son calendarizadas, y sus respectivos kernels son mandados a ejecutar a la GPU, donde se realizan las respectivas operaciones con los índices involucrados. Por último, en la última etapa el GPU manda al host la ubicación de los elementos requeridos. Al igual que en [6]-[7], la latencia de las transacciones de memoria es disfrazada al realizar las etapas 1 y 3 de manera concurrente. Como prueba, los autores implementaron Mega-KV en una computadora comercial con dos CPUs y dos GPUs, donde reportan que su sistema es capaz de realizar más de 160 millones de operaciones por segundo, lo cual significa haber tenido un factor de aceleramiento entre $1.4\times$ y $2.8\times$ en comparación con otras implementaciones basadas puramente en CPU. También se hizo una comparación del tiempo de respuesta de instrucciones de tipo búsqueda, contra el sistema de almacenamiento en memoria por llave-valor MICA, donde se obtuvo una latencia considerablmente mayor, es decir, de 317.4 microsegundos en comparación a 57 microsegundos.

En [16] se plantea utilizar las características de la arquitectura del hardware para generar código eficiente para bases de datos en memoria, para ello implementan *Voodoo*, el cual es un conjunto de operaciones capaces de abstraer las propiedades del hardware con el que se cuenta para así generar código adaptado a tales propiedades. En particular, el compilador de *Voodoo* es portable y genera código tipo OpenCL, el cual es capaz de expresar y utilizar varias optimizaciones a nivel de hardware. Para medir el desempeño de *Voodoo*, se comparó contra Hyper, MonetDB/Ocelot donde la primera funcionan mediante un CPU, y la última es una base de datos híbrida acelerada mediante una GPU. Con respecto a Hyper, se encontró que *Voodoo* generó código más eficiente para busquedas intensivas, pero no para operaciones de ordenamiento. Mientras que en comparación con Ocelot, en casi todas las pruebas el código generado por *Voodoo* fue un poco más ineficiente. Aunque el código generado por *Voodoo* no fue el más eficiente en todos los casos, es interesante la idea de que es portable y adaptable al hardware del equipo, lo que le da una gran versatilidad para distintos ambientes, lo cual es más sencillo de operar que estar diseñando código complicado en específico para una aplicación en particular.

Algo interesante de [14]-[16] es que a diferencia de [5]-[6], no se busca evadir el costo de las transferencias de memoria, aunque en algunos casos como en [14]-[15] se busca esconder, se es consciente de que este costo debe pagarse para realizar una implementación donde coopere el GPU con otros componentes de hardware. También, es interesante la perspectiva que se adopta en [14]-[16], ya que se implementan sistemas híbridos y en algunos de ellos se emplean como parte del sistema las ideas de compresión de [11]-[13]. Otro punto importante es que aunque en [6] y [7] se buscaba evadir el costo de transferencia de memoria, cuando esto no era posible, se realizaba implícitamente una implementación híbrida donde cooperaba tanto el CPU como el GPU.

IV-D. Arquitecturas Integradas

Las implementaciones o estrategías anteriores han lidiado con la latencia asociada a las transferencia de memoria, lo cual ha llevado a que vendedores de hardware como *Nvidia*, *AMD* e *Intel* hayan desarrollado sus propias arquitecturas integradas, es decir, arquitecturas que incluyen tanto un CPU como una GPU en un mismo chip. *Nvidia*, mediante el proyecto *Denver* [17], creó un microprocesador de 64 bits con un conjunto de instrucciones ARMv8-A, el cual, mediante un decodificador a nivel hardware y un recompilador binario a nivel software es capaz de integrar varios coprocesadores, para ello incorporó una memoria compartida entre dispositivos. *AMD* [18], desarrolló el microprocesador *Kaveri*, el cual incluye en su arquitectura una memoria compartida entre el CPU y el GPU. En la misma línea, *Intel* [19], desarrolló las tarjetas gráficas *iGraphics* las cuales incorporó en un mismo chip junto a sus procesadores, lo cual nombró como *Skylake*, y al igual que *AMD*, se cuenta con una memoria compartida entre el CPU y el GPU. Como se mencionó anteriormente, estas implementaciones incorporan una memoria compartida entre dispositivos, de esta manera tanto el CPU como el GPU pueden acceder a las misma estructuras de datos a través de los mismos apuntadores, y aunque así se evita la latencia entre las

transferencias de memoria entre el CPU y la GPU es necesario considerar los siguientes puntos:

Primero, como se menciona en [20], los protocolos para asegurar la consistencia de la memoria son realizados a nivel software mediante librerías especializadas, las cuales, inevitablemente, introducen un overhead durante su ejecución. En particular, en [20] se muestra el desempeño de dos arquitecturas integradas basadas en [17] y en [18], respectivamente. Para ello se ejecutan distintos benchmarks, donde los autores reportan que dependiendo del tipo de aplicación, podría tener menor overhead el considerar las transferencias de memoria entre el GPU y el CPU que considerar una memoria compartida. En la misma línea que en [20], en [21] se menciona que al tener una memoria compartida es necesario contar con protocolos de coherencia de memoria, sobre todo para aplicaciones donde tanto el CPU como el GPU trabajan de manera cooperativa, lo que conlleva a que estas arquitecturas tengan un diseño complejo, y para aprovechar este diseño se requiere de aplicaciones específicamente diseñadas para explotar la cooperación entre ambos dispositivos. Dicho lo anterior, en [21] se proponen 14 aplicaciones para probar el desempeño de dichas arquitecturas. En particular, en [21] se proponen aplicaciones de prueba para realizar cooperativamente particionamiento de datos así como operaciones de grano tanto fino como grueso. Los autores probaron sus aplicaciones utilizando la arquitectura de *Kaveri* contra una arquitectura tradicional de CPU-GPU con transferencias de memoria, donde se reportaron rendimientos superiores de entre un $10\,\%$ y un $82\,\%$ en la arquitectura integrada.

Segundo, como se menciona en [22], otro punto importante a considerar en este tipo de arquitecturas, es que las tarjetas gráficas que utilizan no son tan potentes en relación con una tarjeta gráfica externa, por ello, en [22] se explora si es mayor el beneficio que aporta una arquitectura de este tipo al tener una tarjeta gráfica no tan potente en comparación con la latencia asociada con una tarjeta gráfica externa con mayor potencia. En particular, se contrastan operaciones sobre bases de datos relacionales de ordenamiento, map-reduce, gather, scatter, para una arquitectura acoplada *Skylake* [18] contra una híbrida CPU-GPU, donde los autores reportan que la arquitectura acoplada tiene un aceleramiento de un factor entre $2.2\times$ y $2.6\times$. Aunque los autores reportan un aceleramiento, las pruebas que realizan son sobre bases de datos de a lo más $10M$, lo cual tal vez no sea del todo adecuado si lo que se requiere es exhibir a estas arquitecturas como una opción para el procesamiento masivo de la información, ya que una base de datos no muy grande bien podría ser acelerada por alguno de los enfoques de [5]-[7], y con ello obtener resultados completamente diferentes a los reportados. Por otro lado, en [23] utilizaron una arquitectura acoplada para implementar una aplicación llamada *FineStream*. Esta aplicación fue diseñada con el objetivo de realizar consultas optimizadas tipo SQL mediante el paradigma *stream-processing*. Este paradigma consiste en realizar consultas sobre flujos de información continua, es decir, sobre información cuyos eventos no tienen un fin, como la información que reportan los sensores de temperatura segundo tras segundo, o el flujo de información de series de tiempo que se actualizan minuto tras minuto. Lo interesante de esta aplicación es que, aunque no trabaja sobre una base de datos de un tamaño predeterminado como todas las implementaciones revisadas anteriormente, implicitamente es capaz de procesar una gran cantidad de información que se genera de manera continua, lo que es sumamente valioso para modelos estadísticos de predicción que necesitan reajustarse en intervalos muy cortos de tiempo dada la nueva información que reciben, como los precios pronosticados que se reportan en los mercados financieros, por ejemplo. En particular, en [23] se implementa *FineStream* en la arquitectura *Kaveri* de AMD, y esta es comparada contra *Saber* el cual es una implementación para realizar *stream-processing* en una arquitectura híbrida CPU-GPU. En particular, en [23] se prueban 4 conjuntos de datos que simulan la generación continua de la información con la cual los autores reportan que *FineStream* puede mejorar el rendimiento hasta en un $50\,\%$ a comparación de *Saber*, y además que el consumo energético mejora en un factor de $1.8\times$.

V. Trabajo futuro

Los enfoques revisados en este ensayo exponen que a pesar del overhead asociado a las transferencias de memoria se continua trabajando en cómo implementar aplicaciones con tarjetas gráficas como coprocesadores. A partir de las ideas anteriores se pueden identificar tres vertientes. Por un lado, lo expuesto en [6]-[16] señala que las implementaciones estan convergiendo a una arquitectura híbrida, donde en un futuro tal vez los vendedores de hadware incorporen varios buses de transferencia de memoria, y a su vez se cuente con soporte a nivel software para planificar dichas transferencias. Por otro lado, lo expuesto en [17]-[23], señala que tal vez en un fututo, los vendedores de hadware incluyan un soporte para la coherencia de memoria a nivel hadware, con lo cual podrán tener arquitecturas acopladas con una cooperación más natural y con mayor facilidad de uso. Por último, derivado de las dos ideas anteriores, puede que los vendedores de hadware desarrollen arquitecturas acopladas con GPUs de mayor capacidad con soporte tanto a nivel hadware como software para la coherencia de memoria.

VI. Conclusiones

Hoy en día existen diversas aplicaciones que requieren procesar rapidamente una gran cantidad de información, para ello, y gracias a su alto nivel de paralelismo, se han utilizado las tarjetas gráficas como coprocesadores. La integración de las tarjetas gráficas como coprocesadores no ha sido una tarea trivial a la fecha, esto es por que existe un inevitable overhead entre las transferencias de memoria entre el CPU y el GPU, y el ocultamiento o eliminación de este trae consigo otros factores que deben considerarse. Dicho lo anterior, en este ensayo se han revisado los principales enfoques e implementaciones donde se ha logrado utilizar a las tarjetas gráficas como coprocesadores. En particular, se revisaron cuatro enfoques, donde cada uno de estos elimina o esconde el overhead inherente

a las transferencias de memoria. El primer enfoque, revisado en [5]-[7], pretende eliminar el overhead al cargar la base de datos o parte de esta en la memoria de la GPU, a excepción de [7], este enfoque puede ser eficiente unicamente para bases de datos pequeñas. En el segundo enfoque, revisado en [11]-[13], aunque se esconde el overhead al realizar transiciones de memoria con información comprimida de una manera secuencial, se obtiene un nuevo overhead asociado a los algoritmos de compresión y descompresión, por lo que este enfoque puede ser adecuado cuando el overhead de estos algoritmos es menor al overhead de las transiciones de memoria. En el tercer enfoque, revisado en [14]-[16], se hace explicita la idea de una arquitectura hibrída donde coopera tanto el CPU como el GPU, esta idea permite replantear lo expuesto en [6]-[13] como una componente de una arquitectura híbrida, y con ello generalizar estos enfoques a implementeciones donde el CPU sea una parte más a optimizar dentro del diseño. Por último, en el cuarto enfoque, revisado en [20]-[23], se elimina el overhead de las transiciones de memoria mediante una arquitectura acoplada con memoria compartida. La memoria compartida ocasiona que en este enfoque sea más complicado diseñar aplicaciones que hagan un uso eficiente de los recursos del GPU y del CPU, también con este enfoque surge un nuevo overhead asociado a los protocolos de coherencia de memoria.

REFERENCIAS

1 Written by Jeff Desjardins, F. and editor, "How much data is generated each day?" Apr 2019. [Online]. Available: https://www.weforum.org/agenda/2019/04/how-much-data-is-generated-each-day-cf4bddf29f/

2 Zhong, H., Lieberman, S. A., and Mahlke, S. A., "Extending multicore architectures to exploit hybrid parallelism in single-thread applications," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 25–36.

3 Nagasaka, Y., Matsuoka, S., Azad, A., and Buluç, A., "High-performance sparse matrix-matrix products on intel knl and multicore architectures," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, ser. ICPP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3229710.3229720

4 Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D., Henry, G., Pabst, H., and Heinecke, A., "Anatomy of high-performance deep learning convolutions on simd architectures," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 830–841.

5 Bakkum, P. and Skadron, K., "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: Association for Computing Machinery, 2010, p. 94–103. [Online]. Available: https://doi.org/10.1145/1735688.1735706

6 Rui, R. and Tu, Y.-C., "Fast equi-join algorithms on gpus: Design and implementation," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, ser. SSDBM '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3085504.3085521

7 Sioulas, P., Chrysogelos, P., Karpathiotakis, M., Appuswamy, R., and Ailamaki, A., "Hardware-conscious hash-joins on gpus," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 698–709.

8 Sanders, J. and Kandrot, E., *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

9 Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P., "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 451–460, 01 2010.

10 Gregg, C. and Hazelwood, K., "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 134–144.

11 Fang, W., He, B., and Luo, Q., "Database compression on graphics processors," *Proc. VLDB Endow.*, vol. 3, no. 1–2, p. 670–680, Sep. 2010. [Online]. Available: https://doi.org/10.14778/1920841.1920927

12 Sitaridi, E., Mueller, R., Kaldewey, T., Lohman, G., and Ross, K. A., "Massively-parallel lossless data decompression," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 242–247.

13 Rozenberg, E. and Boncz, P., "Faster across the pcie bus: A gpu library for lightweight decompression: Including support for patched compression schemes," in *Proceedings of the 13th International Workshop on Data Management on New Hardware*, ser. DAMON '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3076113.3076122

14 Pirk, H., Manegold, S., and Kersten, M., "Waste not… efficient co-processing of relational data," in *2014 IEEE 30th International Conference on Data Engineering*, 2014, pp. 508–519.

15 Zhang, K., Wang, K., Yuan, Y., Guo, L., Lee, R., and Zhang, X., "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *Proc. VLDB Endow.*, vol. 8, no. 11, p. 1226–1237, Jul. 2015. [Online]. Available: https://doi.org/10.14778/2809974.2809984

16 Pirk, H., Moll, O., Zaharia, M., and Madden, S., "Voodoo - a vector algebra for portable database performance on modern hardware," *Proc. VLDB Endow.*, vol. 9, no. 14, p. 1707–1718, Oct. 2016. [Online]. Available: https://doi.org/10.14778/3007328.3007336

17 Boggs, D., Brown, G., Tuck, N., and Venkatraman, K. S., "Denver: Nvidia's first 64-bit arm processor," *IEEE Micro*, vol. 35, no. 2, pp. 46–55, 2015.

18 Bouvier, D. and Sander, B., "Applying amd's kaveri apu for heterogeneous computing," in *2014 IEEE Hot Chips 26 Symposium (HCS)*, 2014, pp. 1–42.

19 Doweck, J., Kao, W., Lu, A. K., Mandelblat, J., Rahatekar, A., Rappoport, L., Rotem, E., Yasin, A., and Yoaz, A., "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

20 Dashti, M. and Fedorova, A., "Analyzing memory management methods on integrated cpu-gpu systems," *SIGPLAN Not.*, vol. 52, no. 9, p. 59–69, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3156685.3092256

21 Gómez-Luna, J., Hajj, I. E., Chang, L., García-Floreszx, V., de Gonzalo, S. G., Jablin, T. B., Peña, A. J., and Hwu, W., "Chai: Collaborative heterogeneous applications for integrated-architectures," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 43–54.

22 Ching, E., Egi, N., Mortazavi, M., Cheung, V., and Shi, G., "Unleashing the hidden power of integrated-gpus for database co-processing," in *GI-Jahrestagung*, 2014.

23 Zhang, F., Yang, L., Zhang, S., He, B., Lu, W., and Du, X., "Finestream: Fine-grained window-based stream processing on cpu-gpu integrated architectures," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 633–647. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/zhang-feng