UNA BREVE INTRODUCCIÓN A CUDA PARA VIDEOJUEGOS

Hans Schaa
hansschaadev@gmail.com

Resumen

El poder de las unidades de procesamiento gráfico (GPU) ha permitido grandes avances en el tratamiento de imágenes. Posteriormente con el comienzo de la era de los multiprocesadores y la computación de propósito general en unidades de procesamiento gráfico (GPGPU), los fines a los cuales se han usado las GPU se expanden desde áreas como la salud hasta los videojuegos. En esta última área es cada vez más frecuente el hecho de necesitar estrategias y métodos que permitan un rendimiento estable (60 FPS) durante la sesión de juego para no dañar la experiencia de usuario. Es en estos casos donde herramientas de computación de alto rendimiento (HPC) pueden hacer la diferencia entre un proyecto con experiencia de usuario fluida o una con problemas. Presentamos un escrito introductorio a la computación heterogénea y como esta nos puede ayudar en la optimización de videojuegos.

Palabras clave

Procesamiento, rendimiento, computación heterogénea, videojuegos.

Abstract

The graphics processing unit (GPU) power has allowed great advances in the treatment of images. With the beginning of the era of multiprocessors and general purpose computing in graphics processing units (GPGPU), the purposes for which GPUs have been used range from areas such as healthcare to video games. In this last area, it is increasingly common to need strategies and methods that allow stable performance (60 FPS) during the game session so as not to damage the user experience. It is in these cases where high-performance computing (HPC) tools can make the difference between a project with a smooth user experience or one with problems. We present an introductory paper on heterogeneous computing and how it can help us in video game optimization.

Keywords

Processing, performance, heterogeneous computing, video games.

Ingeniero en Desarrollo de Videojuegos y Realidad Virtual (Universidad de Talca) especializado en "Optimización de videojuegos". Le apasiona el estudio de todas las tecnologías relacionadas a programación, mención especial para el área de computación gráfica y arte generativo.

Introducción

Un tema de investigación que ha tenido mucha atención en los últimos 30 años ha sido el cómo potenciar el rendimiento que nos pueden entregar los dispositivos computacionales. En el año 1980 la generación de computadoras que salían al mercado consistía en CPUs que operaban con relojes internos de 1MHz. Actualmente los procesadores oscilan entre valores de 2GHz y 4GHz, otorgando a los ingenieros nuevos recursos para enfrentar los problemas que para el año 1980 no se podían solucionar. Si bien el aumento de la velocidad del reloj con la que opera la CPU, el uso de instrucciones SIMD (Single instruction multiple data) y el pipelining de las arquitecturas no son las únicas maneras existentes para sacar provecho al hardware, a menudo son estrategias confiables de aplicar paramejorar el rendimiento informático (Wilt, 2013).

No obstante, en las últimas décadas, los fabricantes se han visto forzados a encontrar alternativas a la idea de acelerar el reloj de la CPU. Debido a varias restricciones (energía, calor, tamaño) en la elaboración de circuitos integrados ya no es posible confiar en la estrategia antes mencionada para sacar potencia adicional de las arquitecturas existentes.

Por otro lado, en el mundo de las supercomputadoras ha habido un aumento en el rendimiento de los procesadores, adicionalmente sus fabricantes han obtenido grandes avances en esta métrica al aumentar constantemente el número de procesadores. En el desafío de obtener mayor potencia de procesamiento adicional para computadoras personales, la mejora en las supercomputadoras plantea una nueva estrategia: en vez de solo buscar aumentar el rendimiento de un solo núcleo del procesador ¿por qué no poner más de uno?

Para el año 2005 los fabricantes de computadoras empezaron a ofrecerlas con procesadores dual core en vez de single core. Los nuevos avances tecnológicos dieron lugar a la revolución multinúcleo permitiendo incorporar procesadores con tres, cuatro, seis y ocho núcleos. Actualmente Intel está en desarrollo de procesadores con 56 núcleos el cual por medio de Hyper Threading es capaz de mantener en ejecución 112 hilos, por otra parte, AMD está trabajando en sus nuevas CPUs con arquitectura Zen 3 y desarrollado el diseño de la versión Zen 4 que cuenta con 16 cores.

Por medio de esta investigación bibliográfica se pretende dar a conocer el potencial que tienen las GPU para la ejecución de tareas de propósito general en el área de los videojuegos, otorgando nuevas herramientas al lector para la resolución de operaciones aritméticas complejas, sacando provecho del hardware de las tarjetas de vídeo NVIDIA por medio de la interfaz de programación paralela proporcionada por CUDA.

A continuación, en la sección I describiremos de forma compendiosa la historia que da origen a las GPU de propósito general. Luego, en la sección II describiremos CUDA y cuáles son sus unidades básicas, a continuación, en la sección III se habla sobre computación paralela heterogénea y ponemos énfasis en el potencial de esta. En la sección IV se describen los distintos tipos de bibliotecas existentes como alternativas a CUDA y cómo optimizar procesos paralelizables como es el hacer Tiling para multiplicar dos matrices. Luego en la sección V abordaremos las distintas implementaciones que CUDA ha tenido en el área de simulación de gráficos 3D, efectos para videojuegos, entre otros. Finalmente, en la sección VI planteamos nuestra visión sobre GPGPU, el trabajo futuro e ideas que surgieron en la elaboración de este documento.

I - Breve historia de las GPU

A principios del siglo XXI las GPU fueron creadas para operar sobre los píxeles de un archivo, dando lugar a una mejora en la capacidad que tienen los computadores para trabajar con gráficos. En general, un shader[1] de píxeles usa su posición en la pantalla, junto a información adicional (coordenadas de texturas, colores de entrada, atributos, etc.) para combinarlas y calcular un color final para un píxel en particular. La aritmética

[1] *Programa definido por el usuario diseñado para ejecutarse en alguna etapa de una GPU*

realizada con los atributos de entrada es controlada por los programadores, factor que dio lugar a que los investigadores se dieran cuenta de la flexibilidad de uso que podían tener las GPU. Estos "colores" o "atributos" podrían ser cualquier tipo de dato, y como un color es representado por números, las posibilidades de nuevos usos para las tarjetas de video son muchas. Gracias a esto los programadores podrían programar los shaders de píxeles para ejecutar operaciones matemáticas arbitrarias en estos datos. De esta manera se estaría "engañando" a la GPU para que hiciera tareas distintas a las funciones intrínsecas para las cuales fueron creadas.

Los resultados obtenidos inicialmente parecían prometedores, sin embargo, se encontraron ciertas consideraciones que quitaban libertad al programador, algunas de ellas son:

- Limitaciones de escritura en memoria.
- Difícil depuración.
- Incierto tratamiento de punto flotante.
- Cualquiera que quisiera usar una GPU para realizar cálculos de propósito general necesitaría aprender OpenGL o DirectX ya que estos siguen siendo el único medio por el cual uno puede interactuar con una GPU.

Estas razones hacían que usar las GPU para propósitos generales fuera una tarea poco trivial, lo cual costó su amplia aceptación.

II - CUDA

En el año 2006 NVIDIA presentó la GeForce 8800 GTX la cual estaba construida con la arquitectura CUDA de NVIDIA. Esta arquitectura contenía diversos componentes nuevos construidos específicamente para la programación sobre GPU. El objetivo de esta moderna tarjeta era mitigar muchas de las limitaciones que impedían que las GPU fueran legítimamente útiles para GPGPU (Wilt, 2013).

CUDA es una plataforma de cómputo paralelo de propósito general y un modelo de programación que utiliza el motor de cómputo paralelo en las GPU NVIDIA para solucionar diversos problemas de cálculo complejos de una manera más eficiente (Cheng, Grossman, y McKercher, 2014). CUDA provee dos niveles de API para controlar el dispositivo GPU y organizar hilos:

- CUDA Driver API: proporciona un alto control sobre el uso del dispositivo GPU.
- CUDA Runtime API: API de nivel superior.

Estructura de un programa CUDA típico

Un programa codificado con CUDA consta de 5 pasos principales:

1. Asignar memoria a la GPU.
2. Copiar los datos desde la memoria de la CPU a la GPU.
3. Hacer uso de los núcleos CUDA para ejecutar un cálculo específico.
4. Copiar los datos desde la memoria de la GPU a la memoria CPU.
5. Destruir los datos de la memoria GPU.

En el algoritmo 1 podemos apreciar un ejemplo simple del conocido ejercicio Hello World usando CUDA:

```
1 __global__ void cuda_hello(){
2  printf("Hola mundo desde la GPU!\n");
3 }
4
5 int main() {
6  cuda_hello<<<1,1>>>();
7  return 0;
8 }
```

Algoritmo 1: Hello World CUDA

En la línea 1 podemos observar la palabra reservada global que tiene como objetivo indicar que es una función Kernel. Este tipo de funciones son llamadas desde la CPU para ser ejecutada en la GPU. Posteriormente en la línea 6 se puede apreciar la llamada a cuda_hello desde el host (CPU) colocando la configuración entre los tres chevrones, esta tiene la siguiente notación ((bloques por Grid, Hilos por bloque)), para efectos del ejemplo no necesitamos paralelizar ninguna tarea, así que se colocó un 1 en los dos parámetros.

Unidades básicas de CUDA

La arquitectura CUDA está constituida por tres componentes que ayudan al programador a aprovechar al máximo la capacidad computacional que nos puede brindar la GPU (Ghorpade, Parande, Kulkarni, y Bawaskar, 2012), estas son:

- **Grid:** Grupo de Threads que son ejecutados en un mismo kernel. Los threads no están sincronizados. Cada llamada a CUDA es hecha a través de la Grid.

- **Blocks:** Las Grids están compuestas por blocks, cada uno de estos blocks es una unidad lógica que contiene varios threads y comparten un mismo programa.

- **Threads:** Cada block está constituido por threads, estos son ejecutados en los núcleos de los multiprocesadores. Hasta el año 2019 el máximo de threads por bloque era de 512, actualmente este valor se duplicó admitiendo como máximo 1024 (Wikipedia, 2020b).

En la figura 1 se ilustra lo anteriormente descrito. En ella podemos apreciar los tres componentes básicos y el orden que adoptan. Gracias a la jerarquía que estos componentes poseen, las posibilidades para lograr paralelismo son muchas.

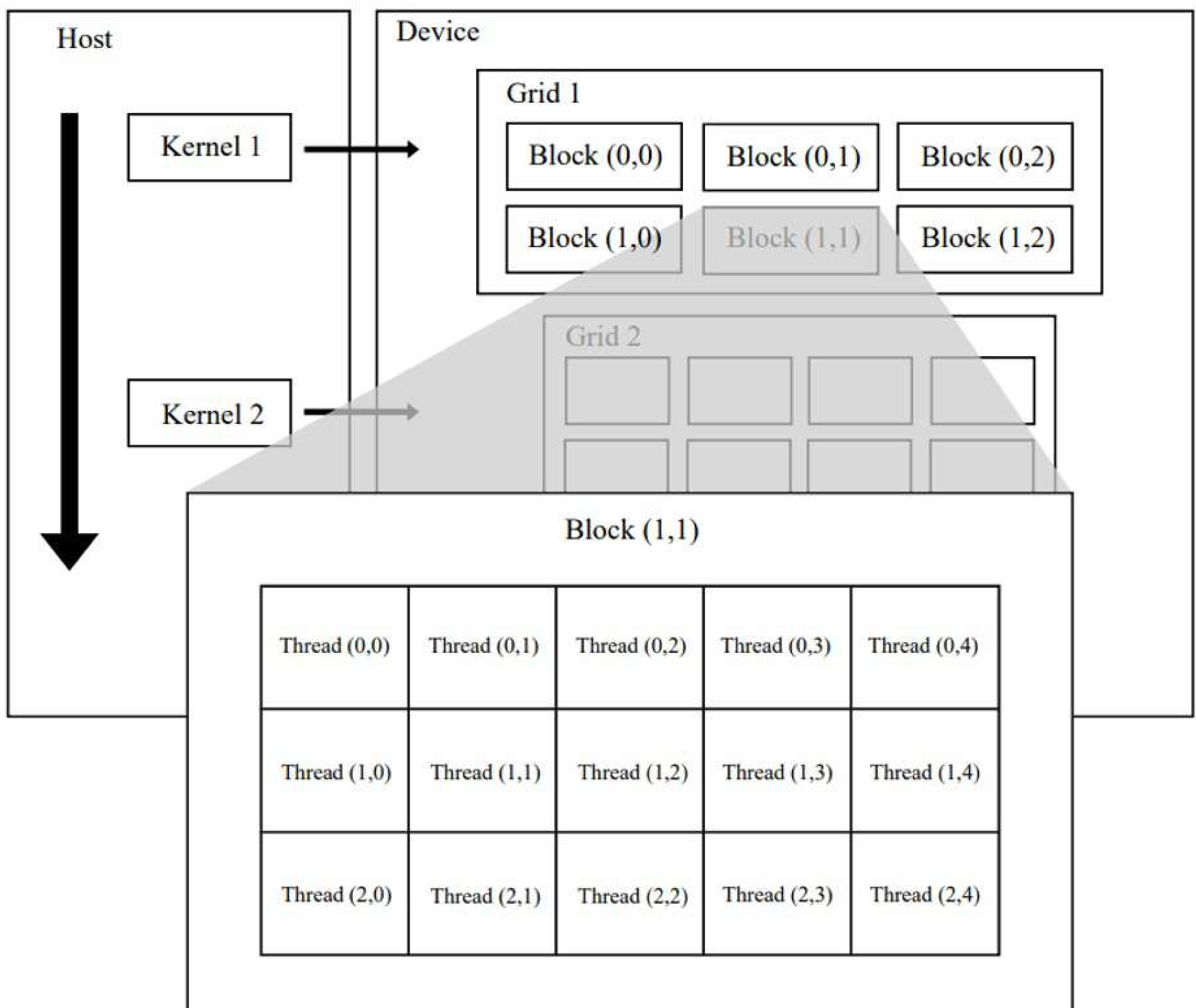

Figura 1: Componentes básicos del hardware queusan las librerías de CUDA (Cheng, Grossman, y McKercher, 2014)

La manera de trabajar obedece a la ejecución de la función Kernel asignada, todos los threads ejecutan el mismo código, esta forma de llevar a cabo las tareas tiene por nombre SPMD (single program, multiple data). Cada thread tiene un índice que usa para calcular las direcciones de memoria que va a acceder y tomar decisiones de control.

III - Computación paralela heterogénea

La informática de alto rendimiento (HPC) está evolucionando continuamente debido a los avances en hardware, por ende, su definición cambia frecuentemente (Cheng y cols., 2014). HPC significa ejecutar una amplia gama de sistemas desde nuestros ordenadores a través de sistemas de procesamiento paralelos de gran tamaño (Dowd y Severance, 2010). Es común pensar que HPC no es solo una arquitectura informática sino también un grupo de elementos (sistemas de hardware, herramientas de software, plataformas de programación y paradigmas de programación paralela). En los últimos años, HPC ha alcanzado un alto desarrollo debido a la aparición de arquitecturas heterogéneas GPU/CPU, que ha provocado un cambio del paradigma fundamental en la programación paralela (Stone, Gohara, y Shi, 2010).

Los arquitectos informáticos, programadores e investigadores se están alejando del constante debate CPU vs GPU (Li, Guo, Shen, y Li, 2020) para concentrarse en la combinación inteligente de estos dos componentes para lograr aún más ganancias computacionales. Este nuevo paradigma conocido como computación heterogénea, tiene como meta hacer que los requisitos de una aplicación encajen con las fortalezas de las arquitecturas CPU/GPU, adicionalmente también obtener un equilibrio en la carga asociada al tiempo de inactividad de ambas unidades de procesamiento (Mittal y Vetter, 2015).

Las arquitecturas que son diferentes de las CPU y GPU presentan una cantidad no menor de desafíos para alcanzar una robusta colaboración entre sus propios componentes. Debido a la interacción entre ellos en un sistema heterogéneo, la optimización del rendimiento y la eficiencia energética requiere tener en cuenta las características de ambas PU (Unidad de procesamiento). Es debido a esta idea, que las técnicas convencionales de optimización solo de CPU o GPU pueden no ser efectivas en un sistema heterogéneo y es necesario implementar novedosas técnicas para alcanzar los objetivos del rendimiento exascale.

Motivación para el uso de Arquitecturas Heterogéneas

Si bien utilizar la GPU de manera independiente parece buena idea, hay varias ideas que motivan el enfoque heterogéneo GPU/CPU. Algunas de las razones que soportan esta idea son:

1. **Las fortalezas arquitectónicas de las PU:** CPU y GPU tienen características arquitectónicas distintas, por un lado, las CPU usan pocos núcleos, estos se ejecutan a una alta frecuencia usando caches de gran tamaño para disminuir la latencia de un solo thread. Estas características hacen a las CPU convenientes para aplicaciones críticas de latencia. Por otro lado, las GPU usan una cantidad mayor de núcleos, los que son en orden y comparten su unidad de control. Otra característica es que hace uso de frecuencias más bajas y caches de menor tamaño (Mittal, 2014), por ende, las GPU son apropiadas para aplicaciones críticas de rendimiento. En resumen, un sistema heterogéneo puede suministrar un alto rendimiento para una variedad mucho más grande de aplicaciones y contexto de uso que el de una CPU o GPU independiente.

2. **Coincidencia de los requisitos algorítmicos con las características de las PU:** En aplicaciones donde la mayor parte del tiempo se hacen transferencias de datos en tiempo de ejecución, o la divergencia de ramificación no permite la ejecución sin interrupciones en todos los núcleos de la GPU, las CPU pueden dar un mejor rendimiento que las GPU. No solo en diferentes aplicaciones, sino que en

las fases de ellas se pueden encontrar propiedades que las hacen más idóneas para ser ejecutadas en una PU particular (Nere, Franey, Hashmi, y Lipasti, 2013; Shen, Arntzen, Varbanescu, Sips, y Simons, s.f.).

3. **Mejor uso de los recursos:** Al momento de cumplir con los requerimientos de rendimiento en el peor de los casos, los sistemas que son solo de CPU o solo de GPU están altamente suministrados de recursos, sin embargo, su uso promedio sigue siendo bajo (Mittal y Vetter, 2015). Adicionalmente, después que la tarea es asignada a la GPU (iniciada en el kernel), la CPU entra en un estado inactivo para ahorrar energía. Análogamente, para aplicaciones con un ancho de banda de memoria de la GPU que actúa como cuello de botella, los recursos usados por la GPU se mantienen altamente utilizados. HPC soluciona estas ineficiencias por medio de la administración inteligente de los recursos para las dos PU. Un hecho que refuerza la idea anterior es demostrado a través del siguiente ejemplo, en la lista Green500 de junio del año 2014 sobre los supercomputadores con un mayor uso eficiente de la energía muestra que los 15 sistemas principales de la lista son sistemas heterogéneos.

4. **Aprovechando los avances en el diseño de CPU:** Los sistemas que tienen GPU, hace un uso convencional de la CPU como host para la GPU, sin embargo, la evolución del rendimiento de la CPU, la han hecho una mejor herramienta para hacer cálculos. Asimismo, muchos trabajos informan que las GPU otorgan una aceleración de hasta 100x a 1000x de speedup, otros investigadores afirman que la aplicación de optimizaciones cuidadosas puede igualar el rendimiento de las GPU (Gummaraju y cols., 2010; Lee y cols., 2010). Es por esto por lo que diversas porciones de divisiones de trabajo para CPU y GPU pueden converger a un rendimiento muy diferente. Los puntos anteriores enfatizan la importancia de usar las capacidades computacionales de la CPU junto a la GPU.

Paralelizando videojuegos para entornos informáticos heterogéneos

Una gran parte del trabajo de GPGPU se ha centrado principalmente en programar algoritmos específicos para trabajar eficientemente en la GPU (operaciones en matrices, procesamiento de imagen y algoritmos de clasificación). Los videojuegos podrían incluir una cantidad alta de cómputos (IA, simulación física, animación de personajes, VFX, render 3D, horneado de sombras en tiempo real, etc.) que pueden sacar provecho de la GPU como procesador de propósito general. Los contextos en los que son ejecutados los videojuegos incluyen al menos una CPU y una GPU en la misma máquina, además de probablemente otros procesadores como es el caso de la SPU en la consola Playstation 3. Algunos de los cálculos hechos en estas aplicaciones se asignan a procesadores con capacidad más idóneas para dicha tarea. Por ende, es importante que se ejecuten cálculos en paralelo en los diversos procesadores de tal forma que se optimice el tiempo de ejecución el rendimiento en general. Es muy útil admitir entornos de procesamiento heterogéneos, pero añade más complejidad a la hora de programar las tareas (AlBahnassi, 2012).

El desarrollo de videojuegos ha ido creciendo en complejidad en la mayoría de los casos. Actualmente pueden incluir una fusión de algoritmos de inteligencia artificial, técnicas de representación 3D, algoritmos de procesamiento de audio y sonido, soporte para juegos en línea, multijugador y muchos más aspectos que hacen la experiencia más atractiva a los sentidos del usuario (Blow, 2004). Las tareas mencionadas anteriormente se ejecutan en forma continua en un bucle de juego, este bucle se repite generalmente en frecuencias de 30 o 60 hertz, en cada bucle se tienen que leer los inputs del usuario, cálculo de físicas, actualización de la lógica de juego para finalmente entregar los resultados en pantalla. Cada bucle de juego tiene que terminar la ejecución en 33.3 o 16.6 milisegundos, dependiendo de si el videojuego tiene como meta 30 o 60 FPS respectivamente. Por esta causa se tuvieron que crear técnicas de optimización para reducir los tiempos de ejecución de los cálculos.

El desarrollo de videojuegos ha ido creciendo en complejidad en la mayoría de los casos. Actualmente pueden incluir una fusión de algoritmos de inteligencia artificial, técnicas de representación 3D, algoritmos de procesamiento de audio y sonido, soporte para juegos en línea, multijugador y muchos más aspectos que hacen la experiencia más atractiva a los sentidos del usuario (Blow, 2004). Las tareas mencionadas anteriormente se ejecutan en forma continua en un bucle de juego, este bucle se repite generalmente en frecuencias de 30 o 60 hertz, en cada bucle se tienen que leer los inputs del usuario, cálculo de físicas, actualización de la lógica de juego para finalmente entregar los resultados en pantalla. Cada bucle de juego tiene que terminar la ejecución en 33.3 o 16.6 milisegundos, dependiendo de si el videojuego tiene como meta 30 o 60 FPS respectivamente. Por esta causa se tuvieron que crear técnicas de optimización para reducir los tiempos de ejecución de los cálculos. Dichas optimizaciones son:

1. Técnicas de partición espacial, como árboles de partición de espacio binario (árboles BSP) y octrees para acelerar las consultas de las entidades del juego (Ericson, 2004).
2. Frustrum Culling para reducir el número de entidades de juego a renderizar.
3. Vectorización de cálculos utilizando intrínsecos del compilador Single-Instruction-Multiple-Data (SIMD) como SSE y AltiVec (Diefendorff, Dubey, Hochsprung y Scale, 2000). Las operaciones de vectores y matrices se optimizan comúnmente con esta técnica.

Si bien estas optimizaciones son útiles para mejorar el rendimiento, no son suficientes en todos los contextos de juego para disminuir el tiempo necesario y que puedan ser ejecutadas en un frame de juego. En la sección V se darán ejemplos de uso de CUDA en videojuegos para solventar ciertos problemas de rendimiento.

IV - Bibliotecas GPGPU

Recientemente GPGPU ha ganado un impulso no menor en la comunidad científica, tal afirmación es respaldada por la gran cantidad de publicaciones relacionadas con GPGPU (Demidov, Ahnert, Rupp y Gottschling, 2013) que se puede consultar en la lista TOP500, debido a esto la creación y mejora de librerías GPGPU tiene un papel importante en la flexibilidad que pueden dar al programador para el manejo y control de la GPU. Hay muchas librerías que permiten su uso, entre ellas las más conocidas son:

1. **CUDA:** acrónimo para Compute Unified Device Architecture desarrollada por NVIDIA, es un motor informático para GPUs NVIDIA al que pueden acceder los programadores por medio de lenguajes de programación como C, Phyton y Java (Yang, Huang, y Lin, 2011). Su arquitectura toma variedades de interfaces computacionales como OpenGL y DirectCompute. Su modelo de programación está diseñado para proveer al ingeniero con intermedios conocimientos de programación una baja curva de aprendizaje baja. En el núcleo de CUDA podemos descubrir tres abstracciones claves: una jerarquía de grupos de threads, memorias que son compartidas y sincronización de barriers, los que están expuestos al programador como un grupo de extensiones del lenguaje.

2. **OpenCL:** Es un nuevo estándar de programación GPU desarrollado por Khronos Group en el año 2008. OpenCL admite varios dispositivos informáticos debido a su nivel de abstracción, no es usado solamente para desarrollar código para la GPU, sino también para la CPU con múltiples núcleos lo que resulta en una herramienta que permite una alta portabilidad (Komatsu y cols., 2010).

3. **SYCL:** Modelo de programación de alto nivel para OpenCL como lenguaje embebido específico de dominio (Wikipedia, 2020a) basado en C++11. Fue desarrollado en 2014 por Khronos Group. SYCL permite que el código para procesadores heterogéneos se escriba en un estilo de fuente única usando C++.

Estas ofrecen muchas facilidades al programador como lo son los IDEs integrados, profilers, debuggers, librerías de alto nivel, etc.

Ejemplo de multiplicación de matrices en CUDA

En esta sección presentaremos el ejercicio de multiplicar dos matrices haciendo uso de la GPU para el cálculo a través de CUDA con la técnica de tiling. Tiling es una técnica muy útil para la optimización de la localidad de datos y se usa en muchas implementaciones que priorizan el rendimiento a la hora de ser ejecutadas, un ejemplo de uso de tiling es en la multiplicación de matrices con alta densidad de datos (Hong, Sukumaran-Rajam, Nisa, Singh, y Sadayappan, 2019). Una táctica común para disminuir el tráfico de memoria es dividir los datos en subconjuntos denominados tiles para que cada uno de estos se ajusten a la memoria compartida, los tiles también pueden ser de gran utilidad cuando los datos son muy grandes para ser almacenados en la memoria global. Como se mencionó anteriormente, tiling mejora el rendimiento del caché, transformando los bucles anidados para que la localidad temporal pueda explotarse mejor para un tamaño de caché específico (Athil, Christian y Reddy, 2014).

A continuación, se mostrará cómo hacer multiplicaciones de matrices con CUDA con tiling. El problema en cuestión viene dado cuando tenemos una matriz A de M x K y una matriz B de K x N, y necesitamos multiplicar A por B para posteriormente almacenar el resultado en una matriz C de M x N. Una implementación en pseudocódigo usando únicamente la CPU sería la que se ve en el algoritmo 2, en él se pueden apreciar la definición de 3 variables matrices donde C[i,j] guarda el resultado de la multiplicación de la fila i por la columna j.

```
1 define A, B, C
2
3 function mult()
4 {
5     for i = 0 to M do
6         for j = 0 to N do
7         /* calcula el elemento C(i,j) */
8             for k = 0 to K do
9                 C(i,j) <= C(i,j) + A(i,k) * B(k,j)
10 }
```

Algoritmo 2: Pseudocódigo multiplicación de matrices

En el algoritmo 3 podemos apreciar la técnica tiling para multiplicación de matrices.

```
1 __global__ void matrixMul_tiling( float* C,
2 float* A, float* B, int wA, int wB)
3 {
4   int bx = blockIdx.x;
5   int by = blockIdx.y;
6
7   int tx = threadIdx.x;
8   int ty = threadIdx.y;
9
10  __shared__ float As[BLOCK_SIZE][
11  BLOCK_SIZE];
12
13
14  __shared__ float Bs[BLOCK_SIZE][
15  BLOCK_SIZE];
16
17  int aBegin = wA * BLOCK_SIZE * by;
18
19  int aEnd = aBegin + wA - 1;
20
21  int aStep = BLOCK_SIZE;
22
23  int bBegin = BLOCK_SIZE * bx;
24  int bStep = BLOCK_SIZE * wB;
25
26  float Csub = 0;
27  for (int a = aBegin, b = bBegin;
28  a <= aEnd;
29  a += aStep, b += bStep) {
30
31  AS(ty, tx) = A[a + wA * ty + tx];
32  BS(tx, ty) = B[b + wB * tx + ty];
33
34
35
36  __syncthreads();
37
38  for (int k = 0; k < BLOCK_SIZE; ++k)
39    Csub += AS(ty, k) * BS(k, tx);
40
41  __syncthreads();
42  }
43
44  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
45  C[c + wB * ty + tx] = Csub;
46 }
```

Algoritmo 3: Multiplicación de matrices con Tiling en la GPU

A continuación, mostramos que ocurre en cada línea:

1. **Línea 1:** La palabra global antes de la declaración de una función indica que esa función es una función kernel que puede ser invocada por el host para crear hilos.

2. **Línea 4 y 5:** Se asignan los índices de los Blocks.

3. **Línea 7 y 8:** Se asignan los índices de los Threads.

4. **Línea 11 y 14:** Declaración de la matriz de memoria compartida. La palabra shared está declarando a As como variable compartida y se le asignará memoria de ese tipo. Así será accesible por un bloque de hilos es decir que todos los hilos de un mismo bloque pueden acceder a ese valor, esta existirá mientras dure la ejecución del kernel, cuando el kernel termina, el contenido de la memoria compartida desaparece. El acceso a la memoria compartida es muy rápido, frecuentemente al programar en CUDA se usa esta memoria para almacenar la de memoria global que es utilizada con una alta frecuencia al momento de ejecutar el kernel.

5. **Línea 17 y 23:** Índice de la primera submatriz de A y B procesada por el bloque respectivamente.

6. **Línea 19:** Índice de la última submatriz de A que será procesada por el bloque.

7. **Línea 21 y 24:** Tamaño de cada bloque para iterar en cada submatriz de A y B respectivamente.

8. **Línea 26:** Csub es usado para almacenar el elemento de la submatriz del bloque que es calculado por el hilo.

9. **Línea 27:** Bucle que recorre todas las submatrices de A y B necesarias para calcular la submatriz del bloque.

10. **Línea 32 y 32:** Carga los elementos de las matrices de la memoria del dispositivo a la memoria compartida; cada thread carga un elemento de cada matriz.

11. **Línea 36:** Sincroniza los hilos para asegurarse que los elementos han sido cargados, cuando un kernel ejecuta syncthreads, todos los hilos de un mismo bloque son retenidos en el punto de llamada hasta que cada uno alcance ese punto. Esto garantiza que todos los hilos de un bloque completen una etapa de ejecución del kernel antes de pasar a la siguiente etapa.

Se pueden encontrar variadas formas de multiplicar matrices incluyendo la expuesta en este documento (Ye, 2020).

V - CUDA en videojuegos

CUDA en videojuegos no es un área muy extendida, por ello a continuación se da a conocer una lista de los estudios que reflejan el uso de CUDA en este ámbito.

Introducción a la prevención de la congestión en la simulación colectiva basada en CUDA

Simular grandes multitudes de personas a una tasa de frames estable se vuelve cada vez más importante en los videojuegos (Bender, Erleben, y Galin, s.f.). Las multitudes de personas colaboran sustancialmente en generar una experiencia más inmersiva. El problema de simular muchas entidades hace de este desafío susceptible a ser resuelto por medio de paralelización. Sacar ventaja del estado actual de la GPU para el cálculo paralelo suena una buena idea, en (Bender y cols., s.f.) se implementa CUDA para desarrollar el concepto de evitar la congestión en el simulador de multitudes generando comportamientos más fieles a la realidad.

El enfoque adoptado se basa en un modelo de fuerza social. La idea es que el objetivo del individuo y la situación que lo rodea generan varias fuerzas sobre el individuo simulado resultando en aceleración y por ende movimiento. La primera publicación más destacada es el artículo de Reynolds (Reynolds, 1987) sobre el comportamiento de las bandadas. El enfoque seguido está más orientado hacia el modelo de fuerza social de Helbing (Helbing y Molnar, 1995).

CUDA ya se ha ocupado para dar solución a estos problemas de simulación de peatones y multitudes. Algunas de las implementaciones previas se pueden encontrar en los artículos de Strippgen y Nagel (Strippgen y Nagel, 2009) que implementaron un enfoque de bases de cola para la simulación. Otras publicaciones se basan en modelos de fuerza social, Richmond et al. (Richmond, Coakley, y Romano, 2009) implementa CUDA para su simulación de peatones, el cual resulta en un problema de complejidad cuadrática, sin embargo, el algoritmo hecho en CUDA se adapta a sí mismo para paralelizar de manera eficiente. En el artículo predecesor (Richmond y Romano, 2008), se usaron fragment y píxel shaders para solucionar el problema.

Aplicación de GPGPU a tiempo real a problemas de IA en juegos

En el artículo (Blewitt, Ushaw, y Morgan, 2013) se tienen en cuenta las razones comerciales y prácticas que obstaculizan el uso de soluciones GPGPU dentro de la IA de los videojuegos, también se revisan los desarrollos GPGPU relevantes en las prácticas más frecuentes de IA dentro de la industria de los videojuegos, para efectos del presente manuscrito detallaremos esta última idea.

Los obstáculos que enfrentan quienes plantean una tarjeta aceleradora de IA, en el sentido de requerir otra pieza de hardware, ya no se dan cuando los ciclos de la GPU pueden ser usados para el mismo propósito. Sin embargo, debemos ser cuidadosos de las restricciones técnicas que conllevan el uso de la GPU para procesar algoritmos de IA en un videojuego, algunas de estas son:

1. Frecuentemente diversos núcleos comparten recursos de caché relativamente pequeños. Una implementación ingenua de algoritmos en la GPU puede conducir a cuellos de botella al momento de acceder a esta memoria.

2. Hacer GPGPU comúnmente requiere operaciones de copia extensivas, tanto hacia como desde la memoria gráfica del sistema, estos son los casos en lo que se espera que la CPU proceda sobre los resultados generados por la GPU, esta sobrecarga a menudo no es trivial.

Estas son algunas de los aspectos con los cuales prestar atención, no obstante, el software actualmente avanza rápido para tratar con estos inconvenientes (Nvidia, 2012).

Al momento de incorporar GPGPU en títulos comerciales de videojuegos, surgen preocupaciones relacionadas con el desvío de los recursos gráficos destinados a procesos de render. En términos generales los recursos asignados a IA son menores que a recursos gráficos, desviar continuamente los recursos dedicados a gráfica produciría un cambio de paradigma significativo. Otra preocupación en el contexto de la IA basada en GPU es que frecuentemente las ganancias de rendimiento que se desprenden de acelerar la GPU en un algoritmo dado se logran con una cantidad de datos muy alta, cantidad de agentes o entidades que extrañamente se ven en los videojuegos. Algunos ejemplos de usos de GPGPU son relatados a continuación.

- **Pathfinding y navegación calculada en la GPU:** Gran parte de los videojuegos necesitan una forma de realizar IA de navegación, dicha tarea puede ser tan compleja como una persecución policial con obstáculos en la ruta o tan simple como un movimiento lineal bajo dos dimensiones. Es inevitable referirse a A* si de pathfinding se trata, las optimizaciones de A* son habitualmente un tema de estudio para plantear mejores algoritmos resolutivos de rutas entre dos puntos, Bleiweiss (Bleiweiss, 2008) presentó una solución GPGPU inspirada en CUDA para la búsqueda de rutas por medio de A*, sus conclusiones indican una aceleración de un orden de magnitud.

  Silva et al. (A. Silva, Rocha, Santos, Ramalho y Teichrieb, 2011) desarrolló una segunda versión del trabajo de Bleiweiss con su propia evolución de A* usando CUDA, logrando manejar mayor cantidad de nodos superando los 340 de Bleiweiss a un valor de 2025 nodos con 65.536 agentes, este observa que existe una restricción limitante y esta es la memoria disponible en la GPU. En 2006 Zioma (Blewitt y cols., 2013) expuso la solución de búsqueda que asignaba una mayor carga de trabajo a la GPU, estudió implementaciones haciendo uso de shaders, describiendo el proceso de administración de datos como texturas a través de DirectX. Los resultados dieron lugar a modestas aceleraciones en el contexto de la implementación Dijstra en GPU por sobre la de CPU, adicionalmente la implementación del algoritmo Floyd-Warshall fue en algunos casos 100 veces más rápida que la de Dijstra en CPU. Otro caso ocurre con el articulo desarrollado por Zhou y Tan (Zhou y Tan, 2009) publicando su implementación de Pathfinding multitudes con múltiples

objetivos logrando una aceleración de más de 11 veces respecto de la implementación en CPU.

En (Blewitt y cols., 2013) se pueden encontrar más ejemplos en el ámbito de Pathfinding con resultados prometedores que sin duda reflejan el potencial de usar GPGPU para este tipo de tareas.

- **Línea de visión y detección ejecutados en la GPU:** Determinar la línea de visión de un enemigo es una característica muy recurrente en videojuegos del género shooter, la optimización de estas tareas puede reducir ampliamente el tiempo de ejecución de las operaciones matemáticas que conlleva el realizar esta labor. Manocha (Manocha, 2005) en el año 2005 aprovechó los beneficios de hacer GPGPU sobre algoritmos geométricos, como por ejemplo determinar la línea de visión de una entidad cualquiera. Observó mejorar en más de un orden de magnitud sobre las actuales soluciones basadas en el uso de la tarjeta gráfica. En este mismo aspecto Perumalla (Perumalla, 2006) pone en práctica el uso de GPU para cálculos de línea de visión para simulaciones de campo de batalla. Los cálculos de la línea de visión directa a la GPU proporcionaron un aumento en tiempo real del rendimiento de 20 veces, lo cual es muy satisfactorio ya que los cálculos para lograr detectar la línea de visión de muchas entidades a la vez pueden llegar a ser altamente caras dentro de la simulación.

  Silva et al. (A. R. D. Silva, Lages y Chaimowicz, 2010) luego de su trabajo con la auto-oclusión en 2009 publicó una extensión de este que presenta una consideración más detallada de la visión que tiene el agente, sus resultados fortalecen las conclusiones anteriormente comentadas respecto de las mejores de rendimiento al usar la GPU.

- **Algoritmos de búsqueda en la GPU:** Los algoritmos de búsqueda son un recurso ampliamente usado en el desarrollo de videojuegos. Kruger et al. (Kruger, Maitre, Jiménez, Baumes y Collet, 2010) publicó en 2010 su investigación referente a un algoritmo de búsqueda local híbrido genérico desarrollado en la plataforma CUDA para la GPU, en comparación con una implementación de CPU de la misma tarea varió entre 47 y 95 veces. Sulewski (Edelkamp y Sulewski, 2009) en 2009 desarrollaron búsqueda en el espacio de estado paralelo usando la GPU, aplicando sus estudios en juegos como Cubo Rubik's y Sliding-Tile Puzzle, tal investigación resultó en una aceleración de 30 veces en contraste de la implementación de CPU con los mismos parámetros de búsqueda.

  Para el año 2010, Bleiweiss (Bleiweiss, 2010) publicó su trabajo que abordaba la aplicación de GPGPU con enfoque en los juegos de suma cero (juegos tridimensionales y cruces, Connnect-4 y Othello), aplicando una variante del algoritmo Minimax, sus resultados reflejan que la implementación en CPU supera a la de GPU hasta que se generan 4000 juegos simultáneamente, con lo cual la naturaleza paralelizada de la GPU demuestra una notable aceleración a 16 000 coincidencias en paralelo.

- **Máquinas de estados finitos en la GPU:** Las FSM son ampliamente usadas en el desarrollo de videojuegos, como por ejemplo en el videojuego Pacman los fantasmas están siendo controlados por medio de máquinas de estados finitos (Thompson, McMillan, Levine, y Andrew, 2008). El desarrollo de máquinas de estados finitos ha sido investigado incluso antes de la llegada de plataformas que facilitan las tareas GPGPU como OpenCL y CUDA. Para el año 2005 Rudomín et al. (Rudomín, Millán, y Hernández, 2005) dio a conocer su implementación de FSM usando shaders a través de GLSL, si bien esta investigación tenía el propósito ser comparada con implementaciones tradicionales, demostró la viabilidad de usar la GPU para la implementación de FSM, más tarde en 2008 Goyal et al. (Goyal, Ormont, Smith, Sankaralingam, y Estan, 2008) generó FSM basadas solamente en GPGPU demostrando una aceleración de hasta 9 veces en relación la implementación en CPU.

Aceleración del texturizado virtual con CUDA

Durante muchos años los videojuegos han usado texturas para mejorar la experiencia del jugador por medio de detalles en las superficies de los objetos y entidades inmersas en estos mundos virtuales (Hollemeersch, Pieters, Lambert, y Van de Walle, 2010). Hay una gran variedad de tipos de texturas que junto a las opciones de luz provocan que los objetos simulan diferentes propiedades como pueden ser la refracción, emisión, rugosidad, etc. Actualmente las exigencias de la comunidad frente a la calidad gráfica de los videojuegos obligan que los ambientes sean más fieles a la realidad, lo que resulta en variados cambios en tiempo real dentro de la escena. últimamente el interés por las tecnologías de texturizado virtual ha ido en crecimiento (Lefebvre, Darbon, y Neyret, 2004), esta técnica permite aplicar texturas del orden de un gigapixel a la geometría de juegos sin exceder las restricciones del hardware actual.

La primera empresa en aplicar esta tecnología de manera comercial fue la empresa ID Software (van Waveren, 2008) usándola en su juego Enemy Territory Quake Wars. Esta tecnología se basa libremente en Clipmapping, la cual tienen la restricción de que solo se permite un rectángulo en la textura con la resolución más alta. Por ende, su uso se restringe casi a geometrías planas con uvs rectangulares. El emplear esta técnica en cualquier tipo de geometría añade una sobrecarga sustancial al método. Los sistemas actuales generan una alta cantidad de trabajo en la CPU y transferencias de datos desde la CPU a la GPU. Van de Walle et. al (Hollemeersch y cols., 2010) plantean el uso de CUDA para transmitir datos entre la memoria del sistema y la GPU de manera eficiente y superar las restricciones existentes en el uso del texturizado virtual.

En esta sección se ha demostrado que los enfoques GPGPU son muy variados en el área de videojuegos, poniendo a disposición del programador un conjunto de herramientas a explotar para ir en beneficio del rendimiento del proyecto. Hay muchas más implementaciones GPGPU en la industria de los videojuegos y experimentos que eventualmente darán lugar a nuevos métodos posibles de usar comercialmente, como es en el caso del artículo de (Yılmaz, Molla, Yıldız y Is¸ler, 2011), donde hacen uso de CUDA para modelar el público de sus escenarios de soccer o en (Amarasinghe y Parberry, 2011) en el que deforman modelos de polígonos para generar combustiones realistas haciendo un bajo uso de la CPU y GPU por medio de CUDA.

VI - Conclusión

Sin lugar a duda la creación de CUDA, el refinamiento de las GPU, la era de los multiprocesadores y los avances alcanzados en HPC han marcado un antes y un después para el mundo informático. Diversas áreas como los videojuegos, cine, salud, entre otros, han aprovechado estos avances. Creemos que el espectro de posibilidades en los cuales las GPU modernas puedan dar solución son muy variadas, más aún en la industria de los videojuegos donde los usuarios se van haciendo cada vez más críticos a la hora de evaluar un videojuego. Áreas como inteligencia artificial, render de texturas, simulación de congestión en ciudad y VFX se han visto beneficiados del estudio de las GPU y su relación con la CPU en entornos heterogéneos. Cada vez es más recurrente la creación de entornos virtuales extensos, para dar la sensación de un mundo a recorrer sin fin y es acá donde las tecnologías antes mencionadas pueden ser de mucha utilidad.

Con el desarrollo de este manuscrito se logra introducir al lector en el ámbito de la computación de propósito general en unidades de procesamiento gráficos (GPU), colocando énfasis en el alcance que pueden llegar a tener en el área de videojuegos y poniendo a disposición herramientas como la paralelización de tareas para lograr un mejor rendimiento.

Finalmente, el trabajo futuro es muy incierto, las posibilidades son demasiadas debido a la capacidad de cómputo de las GPU y la rápida evolución de los procesadores a nivel de rendimiento. Asimismo, la rápida forma en la que se crean nuevas herramientas y pulen las existentes para facilitar las tareas de codificación al programador dejan entrever un sin fin de nuevos experimentos y aplicaciones.

Referencias

AlBahnassi, W. (2012). *Parallelizing Games for Heterogeneous Computing Environments* (Doctoral dissertation, Concordia University).

Amarasinghe, D., & Parberry, I. (2011, June). Fast, believable real-time rendering of burning low-polygon objects in video games. In Proc. 6th Internat. *North American Conf. on Intelligent Games and Simulation* (GAMEON-NA). EUROSIS (pp. 21-26)

Athil, T., Christian, R., & Reddy, Y. B. (2014, April). Cuda memory techniques for matrix multiplication on quadro 4000. In 2014 *11th International Conference on Information Technology:* New Generations (pp. 419-425). IEEE.

Bleiweiss, A. (2008, June). GPU accelerated pathfinding. *In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (pp. 65-74).

Bleiweiss, A. (2010). Playing zero-sum games on the gpu. *In GPU technology conference.*

Blewitt, W., Ushaw, G., & Morgan, G. (2013). Applicability of gpgpu computing to real-time ai solutions in games. *IEEE Transactions on Computational Intelligence and AI in Game*s, 5(3), 265-275.

Blow, J. (2004). Game Development: Harder Than You Think: Ten or twenty years ago it was all fun and games. *Now it's blood, sweat, and code.*Queue, 1(10), 28-37.

Cheng, J., Grossman, M., y McKercher, T. (2014). *Professional CUDA C programming*. John Wiley & Sons.

Demidov, D., Ahnert, K., Rupp, K., y Gottschling, P. (2013). Programming CUDA and opencl: A case study using modern c++ libraries. *SIAM Journal on Scientific Computing*, 35 (5), C453–C472.

Diefendorff, K., Dubey, P. K., Hochsprung, R., & Scale, H. A. S. H. (2000). AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2), 85-95.

Dowd, K., Severance, C. R., & Loukides, M. K. (1998). *High performance computing.*

Edelkamp, S., & Sulewski, D. (2009). Parallel state space search on the GPU. In Proceedings of the International Symposium on Combinatorial Search.

Ericson, C. (2004). *Real-time collision detection*. CRC Press.

Ghorpade, J., Parande, J., Kulkarni, M., & Bawaskar, A. (2012). GPGPU processing in CUDA architecture. *arXiv preprint arXiv:1202*.4347.

Goyal, N., Ormont, J., Smith, R., Sankaralingam, K., & Estan, C. (2008). *Signature matching in network processing using SIMD/GPU architectures*. University of Wisconsin-Madison Department of Computer Sciences.

Gummaraju, J., Sander, B., Morichetti, L., Gaster, B. R., Houston, M., & Zheng, B. (2010, September). Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. *In 2010 19th International Conference on Parallel Architectures and Compilation Techniques* (PACT) (pp. 205-215). IEEE.

Helbing, D., & Molnar, P. (1995). *Social force model for pedestrian dynamics*. Physical review E, 51(5), 4282.

Hollemeersch, C. F., Pieters, B., Lambert, P., & Van de Walle, R. (2010). *Accelerating Virtual Texturing Using CUDA.*

Hong, C., Sukumaran-Rajam, A., Nisa, I., Singh, K., & Sadayappan, P. (2019, February). Adaptive sparse tiling for sparse matrix multiplication. *In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (pp. 300-314).

Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., & Kobayashi, H. (2010, June). Evaluating performance and portability of OpenCL programs. In *The fifth international workshop on automatic performance tuning* (Vol. 66, p. 1).

Krüger, F., Maitre, O., Jim´enez, S., Baumes, L., y Collet, P. (2010). Speedups between× 70 and× 120 for a generic local search (memetic) algorithm on a single gpgpu chip. In *European conference on the applications of revolutionary computation* (pp. 501–511).

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., ... & Dubey, P. (2010, June). Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture* (pp. 451-460).

Lefebvre, S., Darbon, J., & Neyret, F. (2004). *Unified texture management for arbitrary meshes* (Doctoral dissertation, INRIA).

Li, J., Guo, B., Shen, Y., & Li, D. (2020). Low-power Scheduling Framework for Heterogeneous Architecture under Performance Constraint. *KSII Transactions on Internet and Information Systems* (TIIS), 14(5), 2003-2021.

Manocha, D. (2005). General-purpose computations using graphics processors. *Computer, 38*(8), 85-88.

Mittal, S. (2014). A survey of techniques for managing and leveraging caches in GPUs. *Journal of Circuits, Systems, and Computers*, 23(08), 1430002.

Mittal, S., & Vetter, J. S. (2015). A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys* (CSUR), 47(4), 1-35.

Nere, A., Franey, S., Hashmi, A., & Lipasti, M. (2013). Simulating cortical networks on heterogeneous multi-GPU systems. *Journal of Parallel and Distributed Computing*, 73(7), 953-971.

Nvidia, C. (2012). *Nvidias next generation CUDA compute architecture: Kepler gk110*. Whitepaper (2012).

Perumalla, K. S. (2006, May). Discrete-event execution alternatives on general purpose graphical processing units (GPGPUs). In *20th Workshop on Principles of Advanced and Distributed Simulation* (PADS'06) (pp. 74-81). IEEE.

Reynolds, C. W. (1987, August). Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (pp. 25-34).

Richmond, P., Coakley, S., & Romano, D. M. (2009, May). A high performance agent based modelling framework on graphics card hardware with CUDA. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2* (pp. 1125-1126).

Richmond, P., y Romano, D. (2008). Agent based GPU, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the GPU. In *Proceedings international workshop on supervisualisation.*

Rudomín, I., Millán, E., & Hernández, B. (2005). Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory*, 13(8), 741-751.

Shen, J., Varbanescu, A. L., Sips, H., Arntzen, M., & Simons, D. G. (2013, May). Glinda: a framework for accelerating imbalanced applications on heterogeneous platforms. In *Proceedings of the ACM International Conference on Computing Frontiers* (pp. 1-10).

Silva, A., Rocha, F., Santos, A., Ramalho, G., & Teichrieb, V. (2011, November). GPU pathfinding optimization. In *2011 Brazilian Symposium on Games and Digital Entertainment* (pp. 158-163). IEEE.

Silva, A. R. D., Lages, W. S., & Chaimowicz, L. (2010). Boids that see: Using self-occlusion for simulating large groups on gpus. *Computers in Entertainment* (CIE), 7(4), 1-20.

Stone, J. E., Gohara, D., & Shi, G. (2010). *OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering*, 12(3), 66.

Strippgen, D., & Nagel, K. (2009, March). Using common graphics hardware for multi-agent traffic simulation with CUDA. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques* (pp. 1-8).

Thompson, T., McMillan, L., Levine, J., & Andrew, A. (2008, December). An evaluation of the benefits of look-ahead in Pac-Man. In *2008 IEEE Symposium On Computational Intelligence and Games* (pp. 310-315). IEEE.

Van Waveren, J. M. P. (2008). *Geospatial texture streaming from slow storage devices*.

Wikipedia. (2020a). SYCL — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=SYCL&oldid=964248087. ([Online; accessed 09 August-2020])

Wikipedia. (2020b). Thread block (CUDA programming) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/. ([Online; accessed 09-August-2020])

Wockenfuss, F., & Lürig, C. (2011). *Introducing Congestion Avoidance into CUDA Based Crowd Simulation.*

Wilt, N. (2013). *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education.

Yang, C.-T., Huang, C.-L., y Lin, C.-F. (2011). Hybrid CUDA, OpenMP, and MPI parallel programming on multicore gpu clusters. *Computer Physics Communications*, 182 (1), 266–269.

Ye, Z. (2020). *Matrix multiplication in CUDA*. http://www.es.ele.tue.nl/. ([Online; accessed 07-August-2020])

Yılmaz, E., Molla, E., Yıldız, C., & İşler, V. (2011). Realistic modeling of spectator behavior for soccer videogames with CUDA. *Computers & Graphics*, 35(6), 1063-1069.

Zhou, Y., & Tan, Y. (2009, May). GPU-based parallel particle swarm optimization. In *2009 IEEE Congress on Evolutionary Computation* (pp. 1493-1500). IEEE.