Uso de Ray Tracing Cores para el problema k-nearest neighbours

Trabajo final para el curso Computación de Propósito General en Unidades de Procesamiento Gráfico

Facultad de Ingeniería - UDELAR

Guillermo Toyos
guillermo.toyos@fing.edu.uy

Santiago Olmedo
santiago.olmedo@fing.edu.uy

I. Introducción

El advenimiento de la industria de videojuegos en la década de los 70 generó la necesidad de desarrollar hardware capaz de *acelerar* tareas de procesamiento gráfico. Durante los últimos 50 años, la industria le ha demandado a los fabricantes de hardware rápidos avances tecnológicos para el cómputo de gráficos 3D cada vez más rápidos y con mayor realismo. En particular, en los últimos años he resurgido el interés por el algoritmo de renderización *ray tracing*, una técnica capaz de generar imágenes foto-realistas a un mayor costo computacional que la técnica de *rasterización*, técnica predominante en la gran mayoría de las aplicaciones gráficas de las últimas décadas.

Para posibilitar el uso de técnicas de ray tracing en tiempo real, el fabricante de de GPUs Nvidia ha introducido en sus últimas arquitecturas Turing y Ampere una estructura de hardware llamada RT Cores (Ray tracing cores). Permitiendo calcular como la luz viaja en una escena 3D en un tiempo de cómputo mucho menor en comparación con GPUs que no cuentan con esta tecnología. Así posibilitando el uso de ray tracing para el desarrollo de aplicaciones gráficas interactivas.

Si bien los RT Cores fueron concebidos para acelerar tareas esencialmente gráficas. Resulta relevante analizar la posibilidad de aprovechar este hardware para acelerar el procesamiento de tareas ajenas al renderizado de escenas 3D.

El objetivo de este trabajo es analizar los RT Cores e investigar sobre el estado del arte de su uso para la computación de propósito general. Particularmente nos centraremos en el trabajo de Zhu [8], el cual propone el uso de RT Cores para acelerar un algoritmo de búsqueda KNN.

La sección II introduce todos los conceptos relacionados al algoritmo de ray tracing que van a servir para entender el resto del informe. Luego, la sección III explica nociones de RT Cores en el contexto de la arquitectura de tarjetas gráficas Turing y Volta. La sección IV explica cómo utilizar los RT cores en una búsqueda KNN en un espacio 3D. Seguidamente, la sección V consta de un experimento de la implementación de este algoritmo y se estudia su tiempo de ejecución. Por último, la sección VI presenta algunas conclusiones de la experiencia. Además, se agregó la sección VII que sirve como anexo para entender la implementación de los problemas expuestos. Y de como trabajar con la librería OptiX para implementar problemas de ray tracing.

II. Ray Tracing

El algoritmo de ray tracing es una técnica de renderizado (esto es, una estrategia para generar una imagen a partir de un modelo 2D o 3D) que utiliza el comportamiento natural de la luz para generar imágenes: La luz puede modelarse como rayos, provenientes de distintas fuentes energéticas cómo el sol o una lámpara, que viajan en línea recta a través de un medio hasta impactar con un objeto que interrumpe su trayectoria. [6] Al impactar, este rayo de luz puede sufrir múltiples transformaciones:

1. Absorción: Parte de la energía de la luz es absorbida por el objeto, por lo que los rayos generados a partir del impacto tienen menos intensidad.
2. Reflexión: Parte del rayo se refleja, cambiando su dirección.
3. Refracción: El rayo viaja a través del objeto, cambiando su espectro magnético y longitud de onda.
4. Fluorescencia: El objeto absorbe la energía y vuelve a emitir el rayo con una longitud de onda y dirección distintas.

Estos fenómenos están gobernados por múltiples leyes físicas que escapan el objetivo de este trabajo. La idea clave es que en una escena existen fuentes que generan rayos de luz, estos viajan y se transforman al impactar con los distintos objetos de una escena.

¿Y cómo se genera la imagen? Dada una cámara posicionada arbitrariamente en la escena, los múltiples rayos, con distintos ángulos y longitudes de onda, que finalmente impactan, permiten construir la imagen de la escena. Un algoritmo que construye una imagen utilizando este modelo se la llama algoritmo de ray tracing. [2]

Ahora bien, dado que una fuente genera infinitos rayos que a su vez viajan en la escena durante un tiempo indefinido, resulta computacionalmente inviable calcular todas las trayectorias de los rayos para generar una imagen. Por ello, para implementar un algoritmo de ray tracing se aborda el problema inverso: A partir de la cámara, se emiten una cantidad arbitraria de rayos y se calcula su trayectoria para determinar con los distintos

objetos que impactan para calcular su color e intensidad resultantes. Esta técnica se la conoce como **path tracing**.

La imagen 1 muestra cómo se realiza el path tracing de un rayo. Notar que el plano es la imagen resultante, este plano se divide en pequeñas cuadrículas (llamadas píxeles) y para definir el color de cada píxel se realiza una correspondencia entre los rayos generados desde la cámara con el píxel que atraviesa al pasar por el plano.

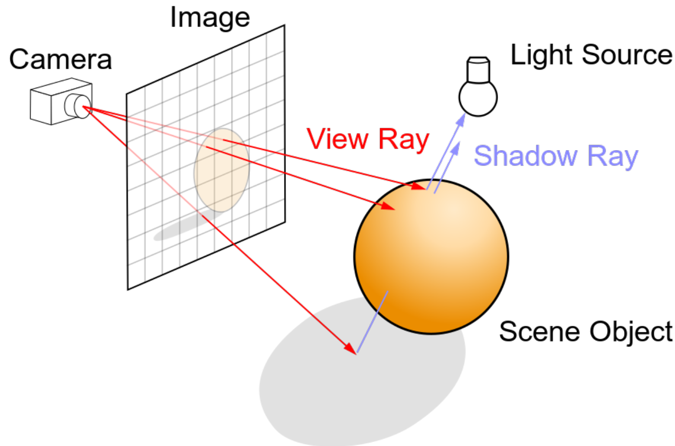

Figura 1: Ilustración de path tracing

Por lo tanto, el algoritmo de path tracing debe determinar para cada rayo con cuales objetos de la geometría de la escena este colisiona. A esto se lo conoce como *ray casting* y es la actividad más costosa del algoritmo (CITA).

II-A. *Bounding volume hierarchy (BVH)*

Dado un rayo $P(t) = dt + o$ donde o es el punto origen del rayo y d un vector de dirección, y un conjunto de volúmenes V, determinar con que volumen colisiona el rayo consiste en hallar el volumen $v \in V$ donde se da una intersección con el mínimo $t \in [0, t_{max}]$, done t_{max} es la distancia máxima del rayo.

La forma *naive* de hallar v es calcular t (si existe) para cada $v \in V$ y quedarse con el volumen con el t asociado más pequeño. [6] Sin embargo, este método es ineficiente para escenas complejas y puede mejorarse utilizando estructuras de datos especializadas como bounding volume hierarchy (BVH) la cual se explicará a continuación.

Dado un conjunto de primitivas geométricas (esferas, triángulos, etc.) es posible organizar estas definiendo un volumen (por ejemplo, un cubo) que contenga a las primitivas geométricas a agrupar. A su vez, estos volúmenes pueden agruparse definiendo un nuevo volumen más grande que los contenga. De esta manera, es posible construir de forma recursiva una jerarquía de volúmenes como se muestra en la figura 2. A esta estructura se la conoce como **bounding volume hierarchy (BVH)**.

Si bien puede utilizarse cualquier volumen para agrupar las primitivas, una opción popular es utilizar prismas (o rectángulos, en escenas 2D) alineados con los ejes del sistema

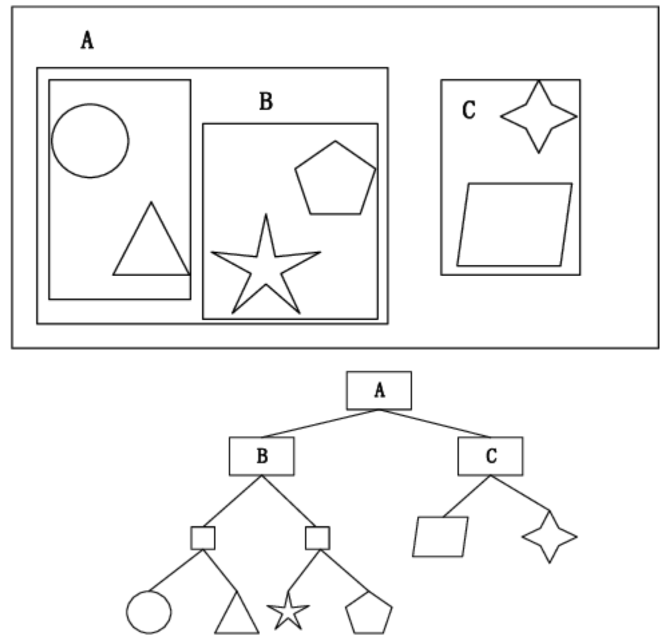

Figura 2: Ejemplo de BVH para una escena 2D.

de coordenadas de la escena. Esto se conoce como **AABB (axis aligned bounding box)**.

Utilizando esta estructura, es posible reducir la cantidad de tests de intersección necesarios para determinar donde impacta el rayo. Por ejemplo, en la figura 2 si se determina que no hay intersección del rayo con el AABB B, no es necesario calcular la intersección con las 4 primitivas geométricas contenidas dentro del AABB B. [**?**]

III. Ray Tracing Cores

En 2018, Nvidia lanza su línea de tarjetas gráficas RTX con una nueva microarquitectura llamada Turing. Esta arquitectura presenta en cada uno de sus Streaming Multiprocessors (SM), un Ray Tracing Core (RT Core) que contiene dos unidades de cómputo especializadas para acelerar las siguientes operaciones:

1. Recorrida de árboles BVH. 2. Testing de intersección rayo/triangulo.

Cada SM delega estas operaciones a su correspondiente RT Core, ahorrándole miles de instrucciones al SM por cada rayo. Esto se traduce, según Nvidia, en un aumento de $10\times$ de performance en comparación con arquitecturas anteriores que no cuentan con RT Cores. [4]

Adicionalmente, la última arquitectura de Nvidia presentada en 2020 (Nvidia Ampere), se le agrega mejoras a los RT Cores permitiéndole al SM realizar otras tareas gráficas o de cómputo general ajenas a ray tracing mientras los RT Cores trabajan en paralelo. Una característica importante es que los RT Cores y los SM comparten la misma memoria, lo cual permite en un mismo kernel, utilizar los RT Cores para ray tracing y los SM para computaciones generales. [5]

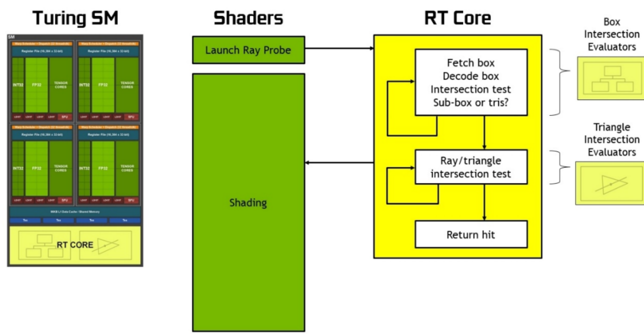

Figura 3: Ray tracing utilizando RT Cores en la arquitectura Nvidia Turing

Para utilizar los RT Cores primero se construye una estructura BVH de la escena y luego se generan rayos. Estos rayos se envían a los RT Cores para obtener la geometría donde colisiona. Para esto, los RT Cores recorren el árbol BVH haciendo tests de intersección de los distintos AABB de la estructura. Además, en el caso de que las primitivas sean triángulos, el test de intersección también es acelerado por los RT Cores. Finalmente, el acelerador le retorna a su SM la geometría donde impacta el rayo para la posterior ejecución de rutinas de cómputo. La figura 3 ilustra como los programas que se ejecutan en el SM (llamados *shaders*, en la jerga de la computación gráfica) utilizan los RT Cores para acelerar la operación de *ray casting*.

En conclusión, las nuevas arquitecturas de GPUs de Nvidia permiten realizar ray tracing sobre múltiples rayos en paralelo utilizando los hilos disponibles en cada SM, donde parte de las operaciones de ray casting se delegan a los RT Cores para acelerar el proceso.

III-A. Programación con RT cores

Para utilizar los RT Cores, Nvidia provee 2 familias de interfaces de programación. Por un lado, existen extensiones para las librerías gráficas Microsoft DirectX y Vulkan. Estas están orientadas para el desarrollo de aplicaciones gráficas. Por otra parte, Nvidia provee una API de ray tracing llamada Nvidia OptiX.

Si bien, estas API permiten construir todo tipo de escenas y shaders con ray tracing, estas tienen un grado de complejidad significativo, particularmente para usuarios sin experiencia en computación gráfica. Por ello, se han realizado esfuerzos por elaborar soluciones que simplifiquen el uso de estas API. Ejemplo de ello es la librería OptiX Wrapping Library (OWL) elaborada por Wald et al.

III-B. Nvidia OptiX

Nvidia OptiX es una API la cual se puede invocar en kernels de CUDA. Esta API sigue el modelo de ejecución Single Instruction Multiple Rays, la cual corresponde al paradigma SIMD. Es decir, cada instrucción es ejecutada en paralelo por múltiples threads con distintos rayos como entrada.

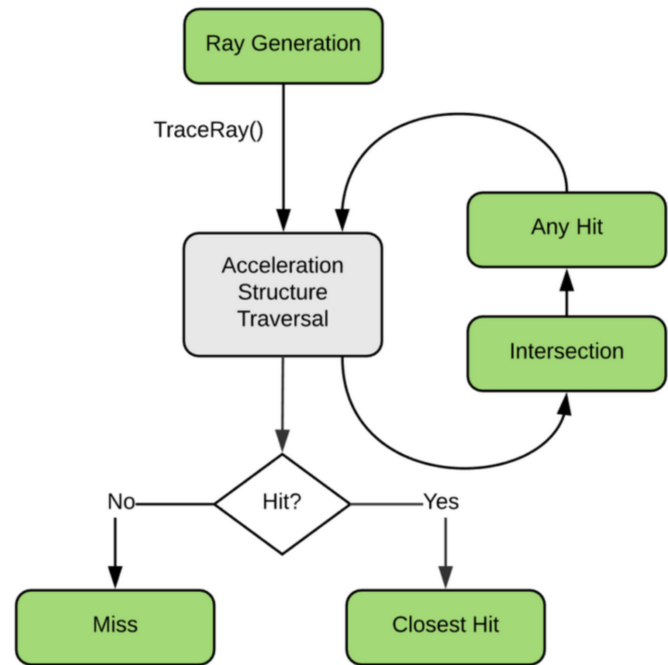

Figura 4: OptiX Ray Tracing pipline. Las rutinas en verde son definidas por el programador

En OptiX, el algoritmo de ray tracing utilizando RT Cores comienza definiendo la escena: se deben definir las primitivas geométricas y los AABBs para poder construir la estructura de aceleración BVH. Una vez definida esta estructura, es posible lanzar el pipeline de ray tracing, el cual se ilustra en la figura 4. Las rutinas básicas que debe definir el programador para implementar el pipeline son las siguientes

1. Ray generation: Se definen cómo se generan los rayos iniciales, especificando su origen y dirección. 2. Intersection: Define como verificar la intersección del rayo con una primitiva geométrica. 3. Any Hit: Se invoca cuando el rayo intercepta con una primitiva geométrica. 4. Closest Hit: Se invoca cuando el rayo colisiona con una primitiva. Es decir, la intersección más cercana al origen del rayo. 5. Miss: Se invoca cuando el rayo no intercepta con ninguna primitiva.

Observar que las rutinas a implementar son esencialmente funciones CUDA, por lo que es posible definir un algoritmo, estructurado por el ray tracing pipeline, utilizando estas rutinas. Por lo tanto, para utilizar los RT Cores para resolver un problema genérico, es necesario formularlo cómo un problema de ray tracing.

IV. K-NEAREST NEIGHBORS (KNN) PROBLEM UTILIZANDO RT CORES

IV-A. Problema a resolver

En forma genérica, el problema KNN consta en: Dado un conjunto de puntos X, un punto $q \in Q \subset X$ y una función de distancia d definida para X y q, hallar el conjunto de k vecinos más cercanos a q en X para cada Q.

En particular, se va a estudiar la versión del problema KNN en $\mathcal{R}^3$ con distancia euclídea. Donde, existe un parámetro adicional $r \in \mathcal{R}$ tal que todos los puntos pertenecientes a K están a una distancia de q menor a r. [1] Notar que este criterio muchas veces se utiliza en la práctica debido a que los puntos que se encuentran muy lejos generalmente no son de utilidad.

Esta versión del problema KNN es de gran interés para problemas ingenieriles. La cual es utilizada para resolver problemas de computación gráfica, para realizar simulaciones físicas, y en estadística para problemas de clasificación y regresión.

IV-B. Mapeo: De KNN a Ray Tracing

Utilizando la propiedad conmutativa de la distancia, la idea clave para formular el problema KNN a ray tracing es, en vez de verificar que puntos se encuentran a una distancia menor a r de q, verificar si q se encuentra a una distancia r de todos los demás puntos. Esto se puede verificar testeando si q se encuentra contenido en las esferas de radio r centradas en los puntos a testear. En la figura 5 se puede observar gráficamente la idea.

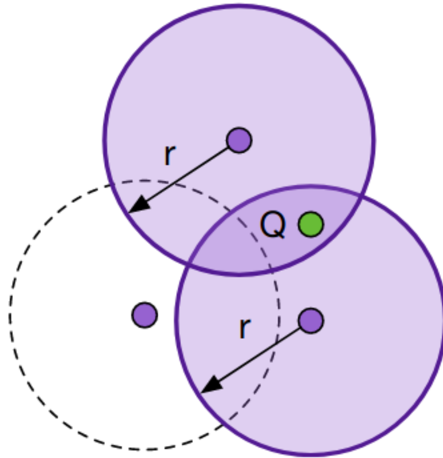

Figura 5: Los vecinos que están a distancia menor a r de q son aquellos que su disco asociado contiene a q. Ilustración obtenida de [8].

Observar que está transformación del problema resulta en la generación de una escena de n esferas. Finalmente, si definimos en rayo épsilon con origen en q, dirección arbitraria y t_{max} infinitesimal. Al realizar ray casting sobre este rayo, las esferas que interceptan con este serán las que se encuentran a una distancia menor a r de q. Por lo tanto, para hallar K basta con calcular la distancia de q con el centro de las esferas que interceptan con el rayo épsilon. De esta manera, el espacio de búsqueda se reduce de todo el conjunto X a los puntos que cumplen con lo anterior.

[1]Notar que puede suceder que el conjunto de puntos a distancia menor a r sea menor a k, en este caso $|K| < k$.

Ahora bien, para utilizar OptiX y los RT Cores es necesario construir una estructura de aceleración BVH y definir AABBs para las esferas. Por lo tanto, para definir la geometría del problema KNN en ray tracing es el siguiente:

1. Para cada punto x en X, se construye un AABB que contenga la esfera/circunferencia de radio r. Luego, se testea si q se encuentra contenido en el AABB. En caso contrario, se desecha el AABB.
2. Si q reside en el AABB, se verifica si q está a una distancia menor que r con respecto a x. Si efectivamente se cumple esto, se lleva una cola de prioridad donde se registra x y su distancia asociada para, al finalizar todas las intersecciones, obtener los k vecinos más cercanos.

También hay que tener en consideración que para hacer la intersección del rayo con los AABBs de la escena se utiliza la estructura de datos de BVH que fue explicada anteriormente.

Si bien el espacio de búsqueda se ve potencialmente reducido, se tiene como contrapartida la necesidad de determinar que esferas contienen al punto q. No obstante, este cálculo se encuentra acelerado por los RT Cores. En virtud de lo anterior, re-formular el problema KNN como uno de ray tracing puede ser ventajoso y resultar en implementaciones que superen los algoritmos KNN del estado del arte, los cuales ya utilizan GPUs con CUDA y estructuras de datos especializadas cómo Octrees [7].

No obstante, debido a que el algoritmo se ejecutará, después de todo, en una arquitectura CUDA. Existen ciertas problemáticas que deberán ser atendidas si se quiere obtener implementaciones eficientes. La siguiente sección expondrá estos problemas y cómo solucionarlos.

IV-C. Optimizaciones

IV-C1. Coherencia de rayos: Debido a que el modelo de ejecución en OptiX es Single Instruction Multiple Rays, en cada CUDA SM, se va a estar ejecutando la misma instrucción para cada rayo. Por otro lado, la recorrida de una estructura arborescente como un BVH, es una actividad altamente divergente en el control de flujo. Por ejemplo, en un mismo SM dos threads pueden recorrer el árbol BVH de forma totalmente distinta, así accediendo a direcciones de memoria distintas y separadas en la memoria. Esto se traduce a un acceso de memoria ineficiente.

Ahora bien, si todos los threads de un SM están trabajando sobre rayos que se encuentran espacialmente adyacentes, la recorrida de la estructura BVH va a ser similar, accediendo a las mismas direcciones de memoria en todos los threads o direcciones cercanas. En estos términos, el acceso a memoria sería coalasced y habría una alta reutilización de las cache L1 y L2.

Por lo tanto, una posible solución para acercarse más al escenario anterior es agrupar los rayos de tal forma que los rayos que se computarán en cada SM sean cercanos.

La interrogante que surge inmediatamente es pensar en que noción de cercanía convendría aplicar. La respuesta es que, consultas en Q que residan en el mismo nodo del árbol BVH (cada nodo que no es una hoja es un AABB), son cercanas.

Es decir, dos consultas son cercanas si están dentro del mismo AABB. Por lo tanto, resulta conveniente construir el primer nivel AABBs tal que agrupen los puntos en Q en particiones con tamaño similar a la cantidad de threads por SM. De tal manera de aprovechar la caché L1 y el acceso coalesced por warp. Esta simplificación ayuda a convertir el problema de planificación de consultas en uno de trazado de rayos.

Para esto, se tendría que trazar un rayo para cada consulta Q y devolver el primer AABB con el que intercepta el rayo; en OptiX, basta con definir el programa de intersección de OptiX para que finalice el pipeline al interceptar con el primer AABB del BVH. Si el rayo intercepta más de un AABB no es relevante, simplemente se necesita relacionar cada consulta con un AABB. Luego para ordenar las consultas dentro del AABB se utiliza el criterio de ordenamiento Morton-Z.

No obstante, [8] confirma experimentalmente que el tamaño de los AABBs es inversamente proporcional a la performance de las recorridas BVH. Por lo tanto, se debe buscar que los AABB que circunscribe a la esfera/circunferencia tiene que ser el más ajustado posible ya que en caso que los AABB sean muy grandes impactará negativamente en la performance de los RT Cores.

Es importante aclarar que si bien OptiX no asegura que los cada rayo se ejecuten en un SM y no se mueva de ahí el autor en [8] asegura que la configuración inicial de los rayos influye significativamente en el impacto de algoritmo./...... escribir bien!

IV-C2. Partición de consultas: Ya se había mencionado previamente que el tamaño de los AABB influye directamente en la performance del algoritmo. Es por ello que en esta sección se va a explicar cómo optimizar el tamaño de los mismos.

La idea es, para cada punto en X, identificar un tamaño de AABB que sea lo suficientemente grande para garantizar la correctitud de la solución. Por otra parte, se buscan formas de reducir el tiempo de las recorridas trabajando con árboles BVH más pequeños. Para ello, para cada una de las particiones de queries construidas en IV-C se construye un BVH optimizado para dicha partición que reduzca la cantidad de recorridas.

Si bien los AABB deben tener por lo menos largo $2r$, es posible utilizar AABBs más pequeños manteniendo la correctitud de la solución siempre cuando los k vecinos más cercanos se encuentren dentro de este AABB.

Para ello, se divide el espacio 2D/3D en celdas y se construye una grilla que contenga a todo el espacio de búsqueda. Luego nos quedamos con la menor grilla posible que contenga los k vecinos (si existen). Este cálculo se hace de manera iterativa y comienza con una celda donde está la consulta q, y aumentando en todas las direcciones hasta que se encuentren k vecinos o hasta que se llegue al radio r. El conjunto de celdas más grande es el cuadrado/cubo inscrito dentro de la circunferencia de radio r. El proceso mencionado se puede ver en la figura 6. A la grilla resultante de este algoritmo, [8] le llama *megacelda*.

Finalmente, solo resta determinar el tamaño del AABB para la consulta q. Para hallar los k vecinos más cercanos a q, se debe hacer un ajuste ya que puede pasar lo que sucede en la figura 7, por lo que sería incorrecto que el AABB sea igual a la *megacelda*.

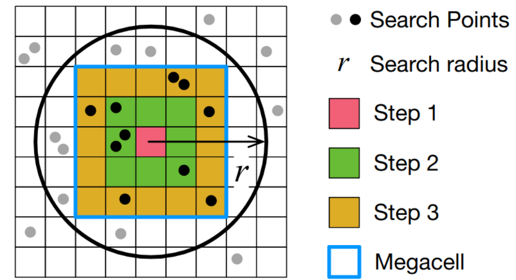

Figura 6: Iteraciones hasta conseguir el AABB. Obtenida de [8].

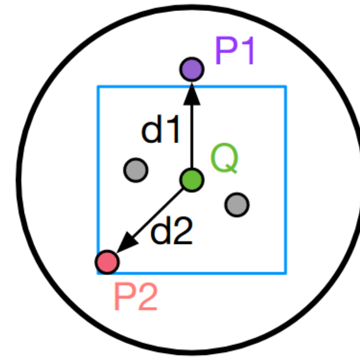

Figura 7: Si bien $P1$ es más cercano a Q que $P2$, debido a que la *megacelda* es cuadrada, el punto está contenido, pero P1 no. Ilustración obtenida de [8].

Por lo tanto, para garantizar la correctitud del algoritmo, se debe tomar la circunferencia que envuelve a la *megacelda* y luego tomar como AABB el mínimo cuadrado que contiene a esta circunferencia. Por lo tanto, basta con multiplicar el largo de la *megacelda* por $\sqrt{2}$ y se obtiene el largo del AABB. En el caso tridimensional, bastaría con multiplicar por $\sqrt{3}$.

V. Evaluación experimental

El objetivo de esta sección es recrear algunas de las pruebas que realiza [8] sobre su implementación original y comparar resultados con el fin de corroborar la performance del algoritmo. Para ello se compara este con un algoritmo del estado del arte que se utiliza actualmente en proyectos de producción sobre datasets reales.

Finalmente, se explora el comportamiento del algoritmo presentado al hacer variar el parámetro r, el cual indica la distancia máxima de los individuos para que sean considerados como vecinos.

V-1. Ambiente: Para las distintas pruebas experimentales se utilizó un ambiente de ejecución provisto por el servicio

Figura 8: Armadillo. Obtenida de [3].

Google Colab. Este provee un ambiente con GPUs para la ejecución de trabajos utilizando notebooks.

Las GPUs utilizadas durante las pruebas fueron Nvidia Tesla T4. Esta GPU de arquitectura Turing, orientada a grandes centros de datos, posee 40 RT Cores y 2560 CUDA Cores.

V-A. *Datos de prueba*

- **Bunny**: este es un modelo creado por Stanford [3]. Y pertenece, junto a Armadillo, a un repositorio con modelos densos en polígonos y son considerados un clásico como datos de ejemplo en el ámbito de la computación gráfica. El conjunto de datos cuenta con un total de 360.000 puntos en R^3
- **Armadillo**: misma idea y fuente que el modelo anterior. La imagen 8 ilustra un renderizado de este modelo utilizando el conjunto de puntos de este dataset. El armadillo se encuentra constituido por un total de $925{,}000$ puntos en R^3.
- **Cosmoflow**: este modelo fue obtenido de [1]. Es un modelo creado a partir de 10000 simulaciones cosmológicas de N-Cuerpos en un espacio tridimensional. Utilizamos 3 versiones de este dataset, uno con 720.000 puntos, otro con 2.5 millones de puntos, y una versión aún más grande con 7.5 millones de puntos.

Es importante destacar que las distintas variantes del dataset CosmoFlow, para contener una mayor cantidad de puntos, se toma un espacio 3D más grande. Por lo tanto, las versiones de CosmoFlow más pequeñas se encuentran contenidas en las versiones con más puntos. Las dimensiones de estas variantes son: $(512, 512, 512)$, de $(1024, 1024, 1024)$, y $(2560, 2560, 2560)$. Donde las mismas contienen 720.000, 2.500.000 y 7.500.000 puntos respectivamente.

Consideramos pertinente utilizar estos dos datasets no solo porque se basan en datos reales sino también porque permitirán ilustrar el algoritmo sobre dos tipos de datos distintos: por un lado, datos donde todos los puntos se encuentran en una región muy reducida del espacio, y por lo tanto la densidad de puntos es mayor en ciertas áreas, y por otra parte en los datasets de CosmoFlow los puntos se encuentran distribuidos en el espacio de manera más homogénea.

V-B. *Implementación*

Se va a utilizar la implementación original de [8] (RTNN), la cual incluye las optimizaciones presentadas en este trabajo. Para comparar el rendimiento de este, se va a utilizar como referencia un algoritmo KNN del estado del arte que se utiliza actualmente en producción: Fixed Radius Nearest Neighbors Search (FRNN). Este algoritmo utiliza CUDA y cuenta con implementaciones en la biblioteca PyTorch para resolver problemas de tipo KNN. Consideramos que este es un buen candidato para realizar la comparación ya que también utiliza distintas optimizaciones como las presentadas en el trabajo utilizando la arquitectura CUDA, por lo que la diferencia central entre las dos implementaciones es el uso de los RT Cores. [7] [7]

Para todas las pruebas se dejó fija la cantidad máxima de vecinos más cercanos a buscar en un valor arbitrario $k = 50$.

V-C. *Resultados y análisis*

V-C1. *Comparación de algoritmos:*

Tiempo de ejecución de RTNN y FRNN.

Dataset	RTNN (ms)	FRNN (ms)	Speedup
Bunny	302	905	2.99×
Armadillo	454	1600	3.52×
Cosmoflow-360K	267	833	3.11×
Cosmoflow-720K	295	1539	5.21×
Cosmoflow-7.5M	833	10084	12.1×

La tabla V-C1 muestra los tiempos de ejecución del algoritmo RTNN y FRNN para los conjuntos de prueba. Para RTNN, se tomaron los tiempos con las optimizaciones y parámetros que dan los mejores resultados. Podemos observar que en todos los casos el tiempo de ejecución de RTNN es significativamente menor. Más aún la mejora es aún mayor para el conjunto cosmoflow-7.5M. Estos resultados coinciden con los realizados en las pruebas experimentales de [8]. Aunque los resultados de Cosmoflow-7.35M fueron excepcionalmente buenos.

V-C2. *Comparación de optimizaciones:*

En esta sección se estudia el impacto de las optimizaciones planteadas en la sección IV-C. Particularmente, se van a hacer pruebas sobre el RTNN en 4 escenarios distintos: con todas las optimizaciones, sin la optimización de particiones, sin la optimización de scheduling y sin las dos optimizaciones. Las imágenes 9 y 10 muestran los resultados del experimento para los distintos datasets.

En la figura 9, podemos observar que para Bunny y Armadillo las optimizaciones resultan esenciales y se traducen a una reducción $500\times$ del tiempo de ejecución para el caso de Armadillo. Sin embargo, no se observa el mismo comportamiento para el dataset de CosmoFlow (10). Estos resultados no se habían reportado en las evaluaciones experimentales de [8]. Creemos que esto se debe a las características inherentes al conjunto de datos. Dado que Bunny y Armadillo son datasets más densos, donde los puntos se encuentran agrupados en un espació pequeño, las técnicas de partitioning que permiten

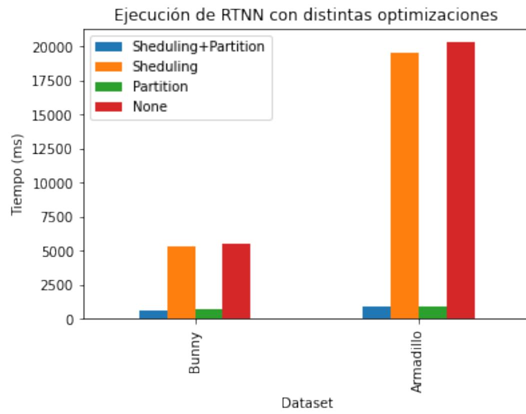

Figura 9: Tiempo de ejecución de Bunny y Armadillo utilizando las distintas optimizaciones.

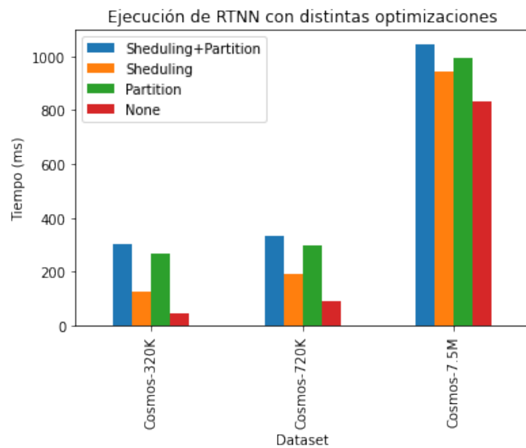

Figura 10: Tiempo de ejecución para las distintas variantes de CosmoFlow utilizando las distintas optimizaciones.

reducir la cantidad de árboles BVH para queries espacialmente cercanas resultan efectivas. Sin embargo, para CosmoFlow donde los puntos se encuentran más dispersos en un espacio más grande, estas técnicas no son tan efectivas y producen un overhead que no reduce el tiempo de búsqueda tanto como en Bunny y Armadillo. En el caso de cosmoflow-320K la diferencia es más marcada ya que el tiempo de búsqueda es mucho menor que para cosmoflow-7.5M, por lo que el overhead generado por las optimizaciones proporcionalmente es mucho mayor.

V-C3. Ejecuciones para distintos valores de r: En las figuras 11 y 12 se pueden ver los tiempos de ejecución para los distintos datasets usando RTNN en función del parámetro de rango de búsqueda máximo r. Para realizar la prueba, se definió un radio l que contenga todos los puntos del conjunto de datos y a partir de este se definieron 25 radios distintos siguiendo la formula $i * l/25 \ : \ i \in [0, 25]$. Por lo tanto, se definieron radios del 1 % al 100 % del radio máximo definido. El objetivo de esto es estudiar cómo se comporta el algoritmo cuando para cada query existe aumenta el número de puntos

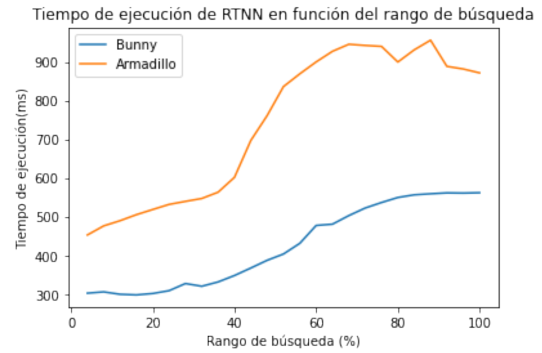

Figura 11: Tiempo de ejecución de RTNN en función del radio de búsqueda máximo.

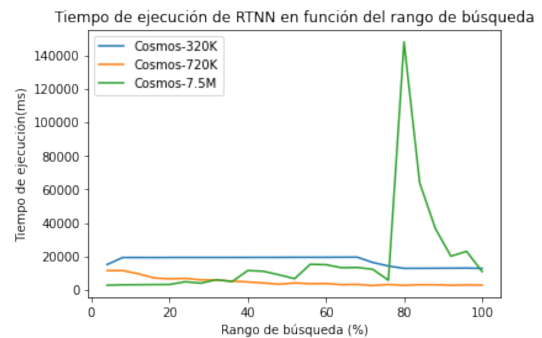

Figura 12: Tiempo de ejecución de RTNN en función del radio de búsqueda máximo.

a considerar. En RTNN esto se traduce a una mayor cantidad de intersecciones a considerar al recorrer la estructura BVH y un tamaño de AABB más grande.

Podemos observar que los datos Bunny y Armadillo tienen una tendencia de crecimiento similar a menos de una constante. Mientras que los datos de CosmoFlow también comparten la misma tendencia de crecimiento entre ellos (a excepción de CosmoFlow-7.5M para r alrededor de 80 % de la escena, que tiene un tiempo de ejecución fuera de la media).

Es interesante observar que para el caso de CosmoFlow el tiempo de ejecución se mantiene casi constante, mientras que en el de Bunny y Armadillo existe un crecimiento gradual que luego se estabiliza después del 60 %. Creemos que esto se debe a que el tiempo de ejecución no es directamente proporcional al rango de búsqueda máximo, pero, como explica [8], en el tamaño de los AABB que podrían generar un número mayor de queries al haber más intersecciones. Pero esto también está ligado a la densidad de los datos. En el caso de CosmoFlow, donde los puntos están en un espacio más esparso, el impacto negativo no es apreciable en el experimento realizado.

VI. Conclusiones

Si bien los resultados de los experimentos realizados son prometedores, aún resta mucho trabajo por realizar. En este informe se presenta un ejemplo de aplicación que puede tomar provecho de los RT Cores, pero aún resta estudiar hasta qué

punto es posible abstraer los distintos conceptos presentados sobre los RT Cores para resolver y acelerar problemas más allá de Ray Tracing. Este informe presenta una aplicación del uso de RT Cores propuesta por [8], y aún no existen muchos más trabajos sobre el uso de RT Cores para acelerar algoritmos de propósito general.

Un aspecto relevante sobre el uso de los RT Cores es que se ejecutan sobre un entorno CUDA. Por lo que varios de los problemas y consideraciones que se deben tener al trabajar sobre CUDA también aplican al transformar problemas para utilizar los RT Cores (particularmente, utilizando OptiX). Debido a ello, consideramos que esta propuesta fue sumamente enriquecedora para el curso y como se vio en las secciones X y X tuvimos la oportunidad de poner en práctica varios de los conceptos vistos en el teórico.

Sin embargo, consideramos que el aprovechamiento de los RT Cores se encuentra severamente limitado por la necesidad de utilizar librerías de Ray Tracing cómo OptiX. Estas introducen un grado de complejidad que no solo genera renuencia, particularmente en investigadores que no tienen conocimientos de computación gráfica, para introducirse a este tema sino también que dificulta la implementación de soluciones ya que se debe tomarse en consideración cuestiones ajenas al problema de Ray Tracing en sí mismo cómo la construcción de la SBT (Shader Binding Table), estructuras de aceleración gráficas y de distintas primitivas geométricas.

Al igual que cuando Nvidia introdujo el modelo de programación CUDA, consideramos que la realización de un esfuerzo para implementar librerías, algoritmos y modelos de programación que simplifiquen el proceso de transformación de un algoritmo a un problema de Ray Tracing sería sumamente beneficioso para la comunidad y potenciaría en gran magnitud el uso de este hardware en la comunidad científica.

VII. Anexo - Programación en OptiX

En esta sección se van a presentar las rutinas y funciones más importantes de la implementación RTNN de [8]. El objetivo es introducir al lector como programar utilizando la librería OptiX y cómo se utilizan sus distintas primitivas para resolver el problema KNN.

Como se vio en la sección III-A, programar utilizando OptiX consiste en definir una serie de rutinas que implementan el pipeline de Ray Tracing. Pero previamente a esto, se deben definir las distintas estructuras de datos de OptiX y la geometría del problema. El código presentado en este anexo sigue esta idea. En la sección sdfG se explica cómo se inicializan todas las estructuras requeridas por el pipeline y en la segunda sección se presentan los *shaders* (rutinas) que se ejecutan en las distintas etapas de este.

VII-A. *Rutinas principales*

El programa principal 21 se ilustran los 2 grandes pasos para implementar la solución. Por un lado, se debe crear un struct que representa el .^{es}tado"de la ejecución del programa, aquí se guardan distintos parámetros, punteros a funciones y arreglos con datos referentes a la ejecución del programa. Esta estrategia es elegida por RTNN y no es un requisito para programar con OptiX.

```
int main(int radius) {
    RTNNState state; //Struct con el estado del programa
    setDevice(state); //Inicializar GPU.
    uploadData(state); //Alocar memoria en la GPU para los puntos

    //setup OptiX
    createContext(state);
    createPipeline(state);
    createSBT(state);

    setupSearch(state);

    //Ejecuci n principal
    createGeometry(state,radius);
    search(state);

    //Liberar Memoria
    cleanupState(state);
    exit(0);
}
```

Código 1: Programa principal

En cambio, la creación del contexto es necesaria por OptiX. Esta puede verse en la figura 13. Primero se define una variable que representa el contexto del device (la GPU). Esta se utiliza para asociar la GPU correspondiente al programa.

```
void createContext( RTNNState& state ) {
    // Initialize CUDA
    CUDA_CHECK( cudaFree( 0 ) );

    // Create Context
    OptixDeviceContext context;
    CUcontext cuCtx = 0;  // zero means take the current context
    OPTIX_CHECK( optixInit() );
    OptixDeviceContextOptions options = {};
    OPTIX_CHECK( optixDeviceContextCreate( cuCtx, &options, &context ) );
    state.context = context;
}
```

Código 2: Inicialización del contexto OptiX

La función optixInit carga la libreria OptiX e inicializa la tabla de funciones que utilizan las diversas rutinas de OptiX. Tras inicializar OptiX se ejecuta la función optixDeviceContextCreate con el parámetro CuCtX=0 la cual crea un contexto de ejecución para utilizar el dispositivo cuda:0 (la primera GPU por defecto).

Una vez creado el contexto es necesario inicializar el ray tracing pipeline como se muestra en 57.

Uno de los pasos más importantes es construir los kernels y asociarlos como módulos de ejecución en OptiX. Para esto, primero se utiliza la función getPtxString para, a partir de un archivo código fuente CUDA, obtener un String que representa las instrucciones en código de ensamblador que utilizara el device. Este código PTX se carga en OptiX utilizando la función optixModuleCreateFromPTX. De esta manera es posible utilizar los kernels CUDA en el pipeline.

Una vez hecho esto, para efectivamente cargar los kernels en el pipeline, primero se debe definir una estructura

que contenga dichos programas. Se utiliza la estructura de OptiX OptixProgramGroup la cual asocia el kernel con el shader que va a implementar del pipeline. Para esto se carga un registro OptixProgramGroupDesc que describe dicho shader, indicado que implementa (por ejemplo, el OPTIX_PROGRAM_GROUP_KIND_MISS el cual corresponde al programa miss explicado en la sección ASDASD, y su kernel asociado, utilizando el atributo entryFuncion. Notar que si no se define una función para uno de los shaders que debe implementar el usuario en el pipeline, se puede definir el puntero de la función como null. Notar que uno de los parámetros que se debe utilizar en la función optixProgramGroupCreate es el contexto definido anteriormente.

```
void createPipeline(RTNNState &state) {
std::vector<OptixProgramGroup> program_groups;

//Opciones para construir el pipeline
state.pipeline_compile_options = {...};

//Se inicializan los modulos donde se encuentran los kernels que implementan las optix shaders.
{
string ptx = getPtxString(OPTIX_SAMPLE_NAME, OPTIX_SAMPLE_DIR, "geometry.cu");
OPTIX_CHECK_LOG(optixModuleCreateFromPTX(state.context, ptx.c_str(),ptx.size(),...));
}{
string ptx = getPtxString( OPTIX_SAMPLE_NAME, OPTIX_SAMPLE_DIR, "camera.cu" );
OPTIX_CHECK_LOG(optixModuleCreateFromPTX(state.context, ptx.c_str(),ptx.size(),...));
}

//Se cargan los shaders
//Ray Generation Shader
OptixProgramGroup cam_prog_group;
OptixProgramGroupDesc cam_prog_group_desc = {};
cam_prog_group_desc.kind = OPTIX_PROGRAM_GROUP_KIND_RAYGEN;
cam_prog_group_desc.raygen.module = state.camera_module;
cam_prog_group_desc.raygen.entryFunctionName = "__raygen__knn";
OPTIX_CHECK_LOG(optixProgramGroupCreate(state.context,&cam_prog_group_desc,&cam_prog_group));
program_groups.push_back(cam_prog_group);
state.raygen_prog_group = cam_prog_group;

//Intersection, AnyHit & Closest Hist Shaders
OptixProgramGroup radiance_sphere_prog_group;
OptixProgramGroupDesc radiance_sphere_prog_group_desc = {};
radiance_sphere_prog_group_desc.hitgroup.moduleIS = state.geometry_module;
radiance_sphere_prog_group_desc.hitgroup.entryFunctionNameIS="__intersection__sphere_knn";
radiance_sphere_prog_group_desc.hitgroup.moduleCH = nullptr;
radiance_sphere_prog_group_desc.hitgroup.entryFunctionNameCH = nullptr;
radiance_sphere_prog_group_desc.hitgroup.moduleAH = state.geometry_module;
radiance_sphere_prog_group_desc.hitgroup.entryFunctionNameAH = "__anyhit__terminateRay";
OPTIX_CHECK_LOG(optixProgramGroupCreate(state.context,&radiance_sphere_prog_group_desc,&radiance_sphere_prog_group));

//Miss Shader
OptixProgramGroup  miss_prog_group;
OptixProgramGroupDesc  miss_prog_group_desc = {};
miss_prog_group_desc.kind  = OPTIX_PROGRAM_GROUP_KIND_MISS;
miss_prog_group_desc.miss.module = nullptr;
miss_prog_group_desc.miss.entryFunctionName  = nullptr;

OPTIX_CHECK_LOG(optixProgramGroupCreate(state.context, &miss_prog_group_desc,&miss_prog_group))
program_groups.push_back(state.radiance_miss_prog_group);

// Link program groups to pipeline
OPTIX_CHECK_LOG(optixPipelineCreate(state.context, program_groups.data(),state.pipeline,&state.pipeline_compile_options, ...))
OptixStackSizes stack_sizes = {};
for( auto& prog_group : program_groups ){
    OPTIX_CHECK( optixUtilAccumulateStackSizes( prog_group, &stack_sizes ) );
}
OPTIX_CHECK(optixUtilComputeStackSizes(&stack_sizes, ...));
OPTIX_CHECK(optixPipelineSetStackSize(state.pipeline,...));
}
```

Código 3: Creación del Ray Tracing Pipeline

Con los programs groups ya creados, estos se asocian efectivamente utilizando la función optixPipelineCreate, la cual requiere del contexto ya provisto, y los distintos ProgramGroups definidos anteriormente. Además de esto se debe definir el tamaño del stack que va a necesitar el pipeline. Afortunadamente existe una función optixUtilComputeStackSizes que permite calcular el tamaño del stack requerido para luego configurarlo con la función optixPipelineSetStackSize.

El único paso que resta para terminar de inicializar el programa es enviar la shader binding table (SBT) al dispositivo. Para esto se utilizan las funciones vistas en el curso para enviar el pipeline al dispositivo: cudaMalloc y cudaMemcpy.

```
CUdeviceptr d_raygen_record;
CUDA_CHECK(cudaMalloc(reinterpret_cast<void**>( &d_raygen_record ),sizeof(RayGenRecord)));
RayGenRecord rg_sbt;
optixSbtRecordPackHeader( state.raygen_prog_group, &rg_sbt );
CUDA_CHECK(cudaMemcpy(reinterpret_cast<void*>( d_raygen_record ),&rg_sbt,sizeof(rg_sbt),
cudaMemcpyHostToDevice) );
state.sbt.raygenRecord = d_raygen_record;
```

Código 4: Creación de la Shader Binding Table (SBT)

Por último, se define la estructura de aceleración y la geometría del problema (figura 17). Se utiliza la primitiva de OptiX para AABB utilizando la función createAABB y se construye la estructura de aceleración invocando la función optixAccelBuild. Esta estructura es la que se utilizará para construir el BVH y toma como parámetro principal un OptixBuildInput el cual pueden ser curvas, triangulos o AABB (cómo en este caso).

```
void createGeometry(RTNNState& state, float radius ) {
```

```
  CUdeviceptr d_aabb = createAABB(state, batch_id,
    radius);
  OptixBuildInput aabb_input = {};
  aabb_input.type =
    OPTIX_BUILD_INPUT_TYPE_CUSTOM_PRIMITIVES;
  aabb_input.customPrimitiveArray.aabbBuffers   = &
    d_aabb;

    OptixAccelBuildOptions accel_options = {
        OPTIX_BUILD_FLAG_ALLOW_COMPACTION,  //
    buildFlags
        OPTIX_BUILD_OPERATION_BUILD         //
    operation
    };

    //Build Graphic acceleration structure
    OPTIX_CHECK( optixAccelComputeMemoryUsage(state.
    context,&accel_options,&aabb_input, 1, &
    gas_buffer_sizes));

    OPTIX_CHECK( optixAccelBuild(state.context,state
    .stream,&accel_options,&aabb_input,...)
}
```

Código 5: Definición de la geometría del problema.

VII-B. OptiX Shaders

Finalmente, las figuras 47, 35 y 5 ilustran las 3 funciones clave para implementar el problema de KNN como uno de Ray Tracing.

La imagen 47 contiene la función de generación de rayos del pipeline. Observar que de forma similar a CUDA, la función optixGetLaunchIndex permite obtener la identificación del rayo a generar, la cual se definió previamente al lanzar el pipeline. Esta función es clave para llevar a cabo el modelo de programación single instruction, multiple rays de OptiX.

Notar que en RTNN se utiliza un struct params qué, similar al RTNNState permite a los shaders acceder a arreglos con información del problema (ubicación de las queries, puntos, lista de vecinos, etc.).

Para construir el rayo épsilon sobre cada punto del problema, se define el largo del rayo con los parámetros $(t_{min}, t_{max}) = (0, 1 \times 10^{-16})$.

Un detalle de implementación interesante es que los punteros del device son de 64 bits, por lo que para definirlos utilizando el tipo de dato int es necesario 2 variables. Para ello OptiX provee una utilidad packPointer que facilita la conversión.

Finalmente, la función más importante del programa de generación de rayos es la función optixTrace. La cual inicia el proceso de ray tracing dado un handle (el contexto donde se ejecuta el programa), los parámetros del rayo (origen y dirección) y una payload.

La payload de un rayo se utiliza para enviar datos desde la rutina de generación de rayos a los demás shaders que se van a invocar durante el pipeline de ray tracing (any-hit, intersection, closest-hit y miss shaders). Esta payload puede ser leída y escrita por estos shaders utilizando las funciones optixGetPayload y optixSetPayload respectivamente. Estas payload tienen un tamaño limitado de no más de 32 enteros de 32 bits. Por lo que una técnica que se utiliza, como en este caso, es codificar punteros en estos ints para leer arreglos que se encuentran en la memoria global del dispositivo.

```
// Kernel CUDA
extern "C" __global__ void __raygen__knn() {

    const uint3 idx = optixGetLaunchIndex();
    unsigned int queryIdx = idx.x;

    float3 ray_origin = params.queries[queryIdx];
    float3 ray_direction = normalize(make_float3(1,
    0, 0));

    //Epsilon ray
    const float tmin = 0.f;
    const float tmax = 1.e-16f;

    // pointers are 64 bits, so need two 32-bit
    integers. optixPathTracing has an example for
    this.
    float min_dists[K];
    unsigned int u0, u1;
    packPointer( min_dists, u0, u1 );

    unsigned int min_idxs[K];
    unsigned int u2, u3;
    packPointer( min_idxs, u2, u3 );

    float max_key;
    unsigned int max_idx;
    unsigned int size = 0;

    optixTrace(
        params.handle,
        ray_origin,
        ray_direction,
        tmin,
        tmax,
        0.0f,
        OptixVisibilityMask( 1 ),
        OPTIX_RAY_FLAG_NONE,
        RAY_TYPE_RADIANCE,
        1,
        RAY_TYPE_RADIANCE,
        reinterpret_cast<unsigned int&>(queryIdx),
        u0, u1, // min_dists
        u2, u3, // min_idxs
        reinterpret_cast<unsigned int&>(max_key),
        reinterpret_cast<unsigned int&>(max_idx),
        reinterpret_cast<unsigned int&>(size)
    );
}
```

Código 6: Ray generation shader

En la imagen 35 se implementa el shader de intersección. Este se utiliza para verificar si efectivamente hay una intersección con las primitivas de la geometría. Notar que la función optixGetPrimitiveIndex permite obtener la identificación de la primitiva (en este caso un punto) sobre el cual se quiere obtener su distancia.

Por otra parte, la función optixGetWorldRayOrigin permite obtener el origen desde donde fue generado el rayo. Al obtener este punto de origen, y las coordenadas del punto donde se quiere obtener su intersección, es posible determinar la distancia que se encuentran los dos puntos para poder implementar el algoritmo KNN. En el caso de haber una intersección se utiliza la función optixReportIntersection para reportar el evento.

```
extern "C" __global__ void __intersection__sphere_knn()
{
  SearchType mode = params.mode;

  unsigned int queryIdx = optixGetPayload_0();
  unsigned int primIdx = optixGetPrimitiveIndex();

  if (mode == NOTEST) { // this implies that this is an initial traversal
    params.frame_buffer[queryIdx * params.limit] = primIdx;
    optixReportIntersection(0,0);
  } else {
    const float3 center = params.points[primIdx];
    const float3 ray_orig = optixGetWorldRayOrigin();
    float3 O = ray_orig - center;
    float sqdist = dot(O, O);

    if ((sqdist > 0) && (sqdist < params.radius * params.radius)) {

      //Evaluar si se debe insertar el elemento en la cola de prioridad.
      //Obtener cola
      const unsigned int u0 = optixGetPayload_1();
      const unsigned int u1 = optixGetPayload_2();
      float* keys = reinterpret_cast<float*>( unpackPointer( u0, u1 ) );
      const unsigned int u2 = optixGetPayload_3();
      const unsigned int u3 = optixGetPayload_4();
      unsigned int* vals = reinterpret_cast<unsigned int*>( unpackPointer( u2, u3 ) );

      if( //Si se debactualizar la lista de K vecinos...){
        optixSetPayload_5( float_as_uint(sqdist) );
        optixSetPayload_6( primIdx );
      }
    }
  }
}
```

Código 7: Intersection shader

Por último, la rutina de miss, que se ejecuta cuando el rayo no intercepta ninguna primitiva, simplemente invoca la funciona optixTerminateRay. La cual causa que el pipeline de ray tracing termine.

```
extern "C" __global__ void __anyhit__terminateRay(){
  optixTerminateRay();
}
```

Código 8: Any Hit pipeline

Referencias

[1] Cosmoflow datasets. https://portal.nersc.gov/project/m3363/.

[2] Peter Shirley Adam Marrs and Ingo Wald, editors. *Ray Tracing Gems II*. Apress, 2021. http://raytracinggems.com/rtg2.

[3] Stanford University Computer Graphics Laboratory. The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/, 2014.

[4] NVIDIA, editor. *NVIDIA TURING GPU ARCHITECTURE*. NVIDIA, 2018.

[5] NVIDIA, editor. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. NVIDIA, 2021.

[6] Peter Shirley. Ray tracing in one weekend-the book series. raytracing.github.io.

[7] Lixin Xue. Fixed radius nn search. https://github.com/lxxue/FRNN.

[8] Yuhao Zhu. RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing. 2022. https://www.cs.rochester.edu/horizon/pubs/ppopp22.pdf.