

Judging a Type by Its Pointer: Short Paper

Patrick Xavier Marquez Choque*
Universidad Católica San Pablo

Jean Carlo Cornejo Cornejo†
Universidad Católica San Pablo

1 Introduction

En este trabajo se plantea mejorar el tiempo de ejecución de funciones virtuales en GPU, siendo esta consecuencia de tomar código polimórfico y exportarlo a GPU. Para ello se plantea el uso de técnicas que permiten dar a conocer el tipo de dato mediante la dirección en la que está ubicada. Todo esto ya que en GPU no se encuentran tanto soluciones, como investigación del tema, a comparación de CPU que si cuenta con soporte para esto.

La programabilidad, es una barrera que dificulta la adopción a mayor escala de programación en GPU, para esto C++ ha añadido soporte para memoria virtual compartida con CPU, así permitiendo mayor portabilidad en cuanto a códigos que usan *multithreading* en CPU, puedan ser ejecutados en GPU sin problema. La mayor diferencia existe en que no se permite objetos con funciones virtuales que estén presentes en la memoria compartida de estos dispositivos. En el caso de CUDA para tener funciones virtuales presentes en el código, mapea las mismas a través de tablas virtuales (*vTable*) y estas contienen punteros a funciones virtuales (*vFunc*) en base a cada tipo. Por ello mismo para llegar al tipo correcto en la *vTable* será necesario hacer consultas en diferentes posiciones de memoria para cada *thread*. A esto se pensó utilizar el método de precarga pero ello implicaría el desperdicio de ancho de banda de la cache en cargar los punteros a *vTable*. [Zhang et al. 2021]

Explicaremos las propuestas del documento, siendo estas:

- Coal: Solución basada en software, especializada en ubicar en memoria objetos del mismo tipo de manera consecutiva.
- TypePointer: Mecanismo en el hardware que utiliza bits reservados sin utilizar para almacenar direcciones virtuales.
- SharedOA: Framework que permite compartir objetos con funciones virtuales entre CPU y GPU, mediante memoria virtual compartida.

2 Programación Orientada a Objetos en GPU

CUDA y OpenCL tienen una gradual evolución en las características relacionadas al código polimórfico, ya que en C++ este polimorfismo se da a través de la Herencia de clases realizando llamadas a funciones virtuales. Se enfocará hacia estas funciones virtuales utilizando CUDA pero como se mencionó anteriormente este soporte es limitado ya que CUDA necesita que los objetos sean instanciados y asignados manualmente a la memoria limitando nuestra capacidad de crear varias instancias de estas clases.

Es necesario un enfoque ya que las soluciones propuestas por los autores muestran la utilización de una estructura llamada *vTable*, se utiliza esto para la asignación dinámica en tiempo de ejecución de la memoria reservada con el compilador de CUDA pero no permite una creación directa de los objetos dinámicos dentro de la propia GPU.

El estado del arte actual de lo que nos ofrece CUDA son accesos de memoria global utilizando una *vTable* donde se muestra el rango del uso de memoria para todas las funciones virtuales, también las operaciones para obtener los puntos a las funciones globales dependiendo de la cantidad de tipos entre nuestros objetos y realizar

las llamadas correspondientes.

Por último es necesario mencionar que para el análisis de los experimentos se está utilizando CUDA 10.1 en una NVIDIA V100 Volta.

3 SharedOA

Para poder desarrollar estas propuestas, como en el caso de *COAL*, era necesario en primera instancia el permitir almacenar funciones virtuales de objetos en memoria compartida entre CPU y GPU, es así como aparece *SharedOA*, que está hecho en código de CUDA, que permite el uso de la nueva función *sharedNew()*. Este método cumple con dos funciones principales:

- Dedicar espacios de memoria consecutivos en base a los tipos de objeto.
- Crear una estructura con los rangos de direcciones de cada tipo en la cual se llamará el rango de la tabla virtual.

4 COAL

Esta solución propone la siguiente estrategia:

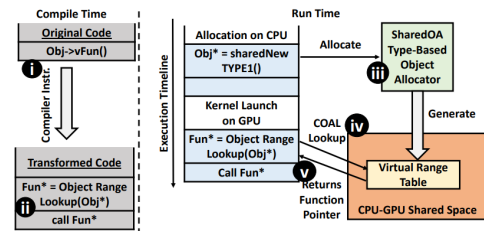


Figure 1: Una vista general de COAL

- En primer lugar nos menciona que *COAL* utilizará tanto la función virtual dentro del objeto heredado del código original como la funcionalidad que realizará el compilador, estas instrucciones serán asignadas mediante un puntero a la función dentro de la tabla donde también estará el puntero al objeto, ordenándolo dentro del rango de todos los objetos de tal manera que será posible retornar correctamente el tipo del objeto que se está utilizando de manera sencilla en lo que se llama como *Virtual Range Table*.
- El principal algoritmo de esta función utilizará un recorrido de un árbol de segmento con complejidad $O(\log_2(K))$ para determinar la posición del rango de todos los objetos y así elegir alguna posición de manera heurística dentro de la tabla de funciones virtuales.
- En relación al tiempo de ejecución para que el código original tenga acceso a nuestra *Virtual Range Table* en primer lugar es necesario una asignación de un puntero al objeto dentro de nuestra memoria principal en CPU y dentro de nuestra función kernel en GPU de tal manera que se genere un espacio compartido a nivel de CPU y GPU que será la tabla mencionada.

5 TypePointer

TypePointer es una alternativa a *COAL*, esta requiere una modificación de hardware, dado que almacena datos en bits sin utilizar y

*e-mail:patrick.marquez@ucsp.edu.pe

†e-mail:jean.cornejo@ucsp.edu.pe

así obtener la ubicación de *vTable* del tipo que sea necesario, este alcance también es necesario aclarar que aunque se realice la modificación de hardware, existe un límite de posibles tipos que pueden usar esta alternativa.

La modificación de hardware proviene de 15 bits no utilizados en el espacio de direcciones virtuales que permiten el uso de 32kb en posible espacio de *vTable*, pero se considera 4k como suficiente para los punteros a funciones virtuales por esto mismo también es necesario la modificación del propio SharedOA para lograr una comunicación entre el hardware y el software para obtener los resultados que buscamos. Dicha estrategia se puede ver de esta manera.

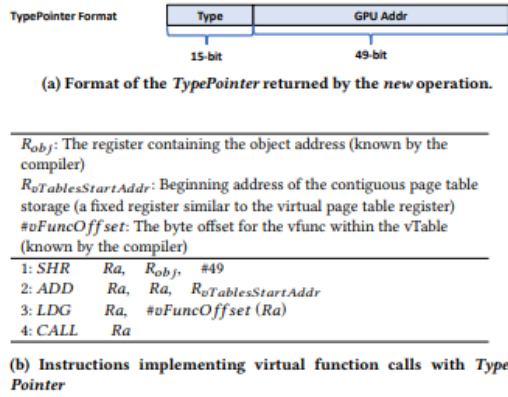


Figure 2: Métodos de Implementación de *TypePointer*.

En la imagen observamos los cambios necesarios a realizar en el compilador para poder utilizar de manera apropiada el registro, mientras que las modificaciones en el hardware se encuentran en un cambio de la lógica del chip MMU.

6 Metodología de los Autores

Como se menciona en la Sección 2 se está utilizando una GPU NVIDIA Volta V100 para realizar todos los experimentos utilizando CUDA 10.1 haciendo una implementación de todas las propuestas de los marcos de trabajo utilizando en la Tabla 3.

En total los autores están utilizando principalmente 3 *Workloads* como espacios de trabajo para realizar los experimentos, el análisis de la experimentación y la comparación de los resultados obtenidos, estos espacios de trabajos son:

- **DynaSOAr:** Es un framework basado en CUDA para la creación de múltiples objetos de simples métodos almacenados en estructuras con una distribución dinámica del manejo de memoria con posibles utilidades para código paralelo. De este espacio de trabajo se está utilizando principalmente ejemplos concretos donde se crean una gran cantidad de objetos simples de distintos tipos con diferentes funcionalidades para realizar ejemplos de simulación como TRAF que es un modelo de simulación del tráfico real, GOL que representa un automata que plantea la herencia genética entre celular simples con su extensión GEN donde se puede apreciar escenarios más complicados de generaciones celulares y por último STUT un modelo que simula la fractura de materiales mediante el método de elementos finitos.
- **GraphChi:** Es un framework para el manejo de algoritmos basados en grafos altamente escalables con diferentes algoritmos como los utilizados dentro de la experimentación que son la clásica búsqueda por anchura o BFS con la definición de las clases de Nodos y Aristas para el recorrido dentro del

grafo, CC que es un algoritmo utilizado para la segmentación de imágenes y por último el algoritmo PR utilizado por algunos motores de búsqueda con actualizaciones frecuentes entre cada nodo para una asignación iterativa entre resultados.

- **Open Source Ray Tracer:** Por último se utilizó una librería simple para la renderización de rayos de luz que entran en acción con objetos como esferas y planos rotando en los mismos con una cantidad limitada de objetos a crear.

Workload	Funcionalidad	Número de Objetos	Número de Tipos	Número de Funciones Virtuales
Dynosoar Workloads				
Traffic (TRAF)	Modelo de simulación del tráfico vehicular	1573714	6	74
Game Of Life (GOL)	Algoritmo de un Automata Celular	5645916	4	29
Structure (STUT)	Simulación de la Fractura de algún material	525000	4	40
Generation (GEN)	Extensión de GOL	1048576	4	33
GraphChi-vE Workloads				
Breadth First Search (BFS)	Algoritmo de Búsqueda en Grafos	2254419	4	5
Connected Components (CC)	Algoritmo de creación de nodos como componentes	2254419	4	6
Page Rank (PR)	Algoritmo de clasificación de páginas en motores de búsqueda	2254419	4	3
GraphChi-vEN Workloads				
Breadth First Search (BFS)	Extensión de BFS	2254419	4	15
Connected Components (CC)	Extensión de CC	2254419	4	15
Page Rank (PR)	Extensión de PR	2254419	4	10
Open Source Ray Tracer Workload				
Raytracing (RAY)	Simulación del renderizado de rayos de luz en objetos	1000	3	3

Figure 3: Workloads utilizados por la experimentación del artículo.

- Para implementar *COAL* se está utilizando la estrategia mencionada en la Sección 4 utilizando un *segment tree* como la estructura responsable de repartir los punteros de los objetos y a las funciones virtuales dentro de nuestra *virtual Range Table*. Con este algoritmo *COAL* se está implementando para los tipos de datos que se van a crear.
- Para implementar *SharedOA* es necesario realizar una modificación de la asignación de memoria compartida por defecto de CUDA, así que se realizó la ejecución de una función kernel de inicialización de la *vTable* de manera dinámica se asignarán un espacio compartido entre los recursos para la CPU y GPU de tal manera que corresponda a la mejora del rendimiento a comparación con la *Virtual Range Table* con una primera ejecución consumiendo aproximadamente un 0.15% del tiempo total de inicialización.
- Por último, para implementar *TypePointer* se realizó la implementación de un prototipo a nivel de *software* necesario porque se está modificando la arquitectura a nivel de registros y la MMU. Se utilizó un *framework* de simulación para modelos de validación de GPU llamado AccelSim basado en SASS donde se evalúan los resultados obtenidos de la utilización de los modelos experimentales de los marcos de trabajo utilizado con y sin la sobrecarga de software para evitar errores dentro de la unidad MMU y para realizar la comparación más adelante.

7 Resultados

Para realizar pruebas en donde se compruebe la efectividad de las técnicas mencionadas en el trabajo es necesario realizar las pruebas

y en este caso para ello tenemos que comparar:

- CUDA.
- Concord.
- SharedOA.
- COAL.
- TypePointer.

En los resultados podemos observar que existe una mejora en el desempeño con los métodos planteados (COAL y TypePointer), tomando en cuenta que todos se dieron ejecutando lo ya mencionado en 6 dado que estás propuestas buscan implementar algo no mencionado ni desarrollado en CUDA.

También cabe resaltar que TypePointer es más eficiente que COAL y que a nivel de instrucciones realizadas existen casos en las que COAL y Concord son los que contienen la mayor cantidad también incluyendo en menor medida a Typepointer, específicamente COAL añade un 83%, Concord un 28% y TypePointer un 19%, especialmente en Concord el caso de esto se da debido a que reduce el número de instrucciones de memoria, pues su memoria tiene un alto índice de *cache-hit*, continuando sobre el manejo de memoria en estos experimentos, en el caso de SharedOA obtiene un 41% mejor desempeño que CUDA y si se le añade COAL se obtiene una mejora de 6%.

Se realizó un estudio de la escalabilidad con cada propuesta, donde el mejor resultado pertenece a TypePointer en relación de la cantidad de tiempo de ejecución con el número de objetos que se está creando.

8 Conclusiones y Trabajos Futuros

En primer lugar es necesario mencionar las fortalezas y debilidades de cada solución propuesta para mencionar las conclusiones y los trabajos futuros.

En primer lugar cabe destacar la experimentación realizada ya que los resultados obtenidos se comparan en CUDA y esto hace difícil la aplicación específica con otros modelos similares ya que tanto Concord y SharedOA están basados en CUDA, esto hace que los resultados de ambas parte en comparación con la utilización de CUDA no tenga mucha diferencia entre la misma.

La propia definición del recurso compartido como la memoria en COAL no es muy eficiente en comparación con SharedOA ya que la complejidad del algoritmo del espacio de trabajo que se experimento se vera afectada de manera significativa por la cantidad de accesos de memoria entre los punteros de los tipos de datos que se están solicitando por lo tanto vemos el bajo rendimiento de COAL en comparación de las demás soluciones tanto al nivel de la cantidad de instrucciones de accesos de memoria como la cantidad de ancho de banda utilizada en relación al tiempo de ejecución global.

Y por último una deficiencia en relación de los tamaños de los datos iniciales con los que se crean estos espacios de memoria al no ser totalmente dinámicos ya que la limitación del espacio compartido necesario tanto para la CPU y la GPU es vital entonces esto hace que se tome tiempo adicional en el nuevo calculo del espacio de memoria que se esta requiriendo duplicando esta cantidad lo que puede llegar a tamaños realmente grandes con un manejo de memoria poco eficiente.

En relación a las resultados donde se puede apreciar que TypePointer es la estrategia más eficiente es debido a la aplicación que se está dando específicamente para este tipo de solución ya que esto, aunque sea modificando tanto a nivel de hardware y software se está analizando con un framework simulado en cuestión de software y

esto no podría representar totalmente la eficiencia de esta solución pero es una muy buena aproximación de la misma.

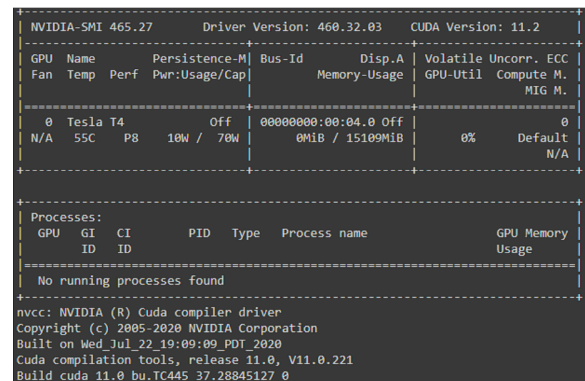
El rendimiento es prueba de esto ya que COAL comparte una mejora desde un 6% a nivel de rendimiento en comparación con la librería por defecto que CUDA proporciona hasta un 80% en aumento del rendimiento del mismo pero que en comparación TypePointer logrará superar hasta un 90% en comparación con la librería CUDA.

Como trabajo Futuro la principal característica de esta experimentación es la utilización de nuestros espacios de trabajos con ciertas características en relación a la herencia para verificar sobretodo la capacidad escalable que tiene la propuesta dada por los autores, esto se realizó en parte en la siguiente sección con ciertas dificultades al momento de realizar la replica de la experimentación realizada en el artículo.

9 Experimentos

En nuestra experimentación utilizamos un alcance al benchmark de los ejemplos propuestos por el paper, a su vez que también tenemos ejemplos simples de ejecución que reflejan el uso de las técnicas explicadas por los investigadores con el objetivo de analizar específicamente las soluciones propuestas por el artículo que son COAL, SharedOA y TypePointer sin mencionar los demás framework utilizados para la comparación.

En primer lugar el alcance de los experimentos utilizados en los diferentes espacios de trabajo se encuentran en nuestro repositorio ¹ pero surgieron bastantes limitaciones al momento de realizar la réplica.



```
NVIDIA-SMI 465.27 Driver Version: 460.32.03 CUDA Version: 11.2
+-----+
| GPU | Name | Persistence-M | Bus-Id | Disp.A | Volatile Uncorr. ECC |
| Fan  | Temp  | Perf         | Pwr:Usage/Cap | Memory-Usage | GPU-Util | Compute M. |
|-----+-----+-----+-----+-----+-----+-----+
| 0    | Tesla T4 | Off         | 00000000:00:04:0 | Off          | 0%       | Default    |
| N/A  | 55C    | P8          | 10W / 70W       | 0MiB / 15109MiB |          |            |
+-----+-----+-----+-----+-----+-----+-----+
Processes:
+-----+
| GPU | GI  | CI  | PID | Type | Process name | GPU Memory |
| ID  | ID  |     |     |      |               | Usage      |
+-----+-----+-----+-----+-----+-----+
| No running processes found |
+-----+
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Wed Jul 22 19:09:09 PDT 2020
Cuda compilation tools, release 11.0, V11.0.221
Build cuda 11.0 bu.TC445 37.28845127_0
```

Figure 4: Especificaciones de Google Colab.

La gran diferencia entre el recurso del Hardware necesario para realizar las pruebas fueron el principal problema para replicar las mismas ya que no se tenía a la disposición ningún computador con características similares a disposición. También el detalle porque algunos de los benchmarks o incluso los experimentos como ejemplos de los espacios de trabajos no estaban disponibles para su uso del todo e incluso el benchmark de la librería Open Source de Ray Tracing había sido retirada de uso debido a ciertos problemas. En relación a esto se realizó una experimentación utilizando el recurso de Google Colab, que se pueden apreciar en la Figura 4, que actualmente tiene a su disposición la implementación de código en CUDA con el que se logró algunos ejemplos simples que están disponibles dentro de nuestro repositorio pero que no fueron totalmente probados y no se concretó los resultados que tuvieron en el artículo.

¹<https://github.com/patrick03524/Programacion-Paralela-y-Distribuida/tree/main/Replica%20Trabajo%20Final>

En el caso de los ejemplos simples tenemos disponibles códigos que estaban preparados para ser ejecutados como lo menciona el trabajo, estos tomaban en cuenta las librerías desarrolladas para ejecutar los *workloads* y junto a ello contenían código que mostraba el uso directo del polimorfismo y funciones virtuales, cada uno mostraba resultados que deberían ser obtenidos en caso de creación de instancias y la ejecución de la misma función, aún presentando ciertas dificultades ya mencionadas, estos ejemplos se pueden ejecutar sin recibir errores, aunque claro es esperado que su comportamiento no sea el mismo al original esperado en donde se ha realizado el cambio de hardware.

References

ZHANG, M., ALAWNEH, A., AND ROGERS, T. G. 2021. Judging a type by its pointer: Optimizing gpu virtual functions. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, New York, NY, USA, ASPLOS 2021, 241–254.