Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA
Unidad Iztapalapa

División de Ciencias Básicas e Ingeniería

POSGRADO EN CIENCIAS Y TECNOLOGÍAS DE LA INFORMACIÓN

GLG: Lenguaje Visual para el Desarrollo de Programas Paralelos Usando GPUs

Presenta

Rodrigo Rivera Cerón

Matrícula: 2212801433

Tesis para obtener el grado de Maestro en Ciencias
(Maestría en Ciencias y Tecnologías de la Información)

Asesores:

Dr. Miguel Alfonso Castro García

Dr. José Luis Quiroz Fabián

Jurado calificador:

Presidente: Dr. Amílcar Meneses Viveros

Secretario: Dr. Ricardo Marcelín Jiménez

Vocal: Dr. Miguel Alfonso Castro García

Iztapalapa, Ciudad de México a 11 de octubre del 2023

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA
Unidad Iztapalapa

División de Ciencias Básicas e Ingeniería

GLG: Lenguaje Visual para el Desarrollo de Programas Paralelos Usando GPUs

Tesis para obtener el grado de Maestro en Ciencias
(Maestría en Ciencias y Tecnologías de la Información)

Presenta
Rodrigo Rivera Cerón
Lic. en Computación

Asesores:

Dr. Miguel Alfonso Castro García
Dr. José Luis Quiroz Fabián

Jurado calificador:

Presidente: Dr. Amílcar Meneses Viveros
Secretario: Dr. Ricardo Marcelín Jiménez
Vocal: Dr. Miguel Alfonso Castro García

Ciudad de México
Octubre 2023

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

ACTA DE EXAMEN DE GRADO

No. 00107
Matrícula: 2212801433

GLG: lenguaje visual para el desarrollo de programas paralelos usando GPUs

En la Ciudad de México, se presentaron a las 10:00 horas del día 11 del mes de octubre del año 2023 en la Unidad Iztapalapa de la Universidad Autónoma Metropolitana, los suscritos miembros del jurado:

DR. AMILCAR MENESES VIVEROS
DR. MIGUEL ALFONSO CASTRO GARCIA
DR. RICARDO MARCELIN JIMENEZ

Bajo la Presidencia del primero y con carácter de Secretario el último, se reunieron para proceder al Examen de Grado cuya denominación aparece al margen, para la obtención del grado de:

MAESTRO EN CIENCIAS (CIENCIAS Y TECNOLOGÍAS DE LA INFORMACIÓN)

DE: RODRIGO RIVERA CERON

y de acuerdo con el artículo 78 fracción III del Reglamento de Estudios Superiores de la Universidad Autónoma Metropolitana, los miembros del jurado resolvieron:

APROBAR

Acto continuo, el presidente del jurado comunicó al interesado el resultado de la evaluación y, en caso aprobatorio, le fue tomada la protesta.

RODRIGO RIVERA CERON
ALUMNO

REVISÓ

MTRA. ROSALIA SERRANO DE LA PAZ
DIRECTORA DE SISTEMAS ESCOLARES

DIRECTOR DE LA DIVISIÓN DE CBI

DR. ROMAN LINARES ROMERO

PRESIDENTE

DR. AMILCAR MENESES VIVEROS

VOCAL

DR. MIGUEL ALFONSO CASTRO GARCIA

SECRETARIO

DR. RICARDO MARCELIN JIMENEZ

Me gustaría dedicar esta tesis a mi yo del futuro. Nunca te rindas, siempre aprenderás algo nuevo cada día y la satisfacción de haber logrado algo que no fue fácil es la mejor de las sensaciones.

Agradecimientos

En primer lugar, quiero agradecer a las personas más importantes para mí: mi hermana Alejandra y mi abuelita Carlota, quienes siempre se preocuparon por mi salud y alimentación.

Agradezco también a mi primo Ian, a mi maestro y amigo Bernardo, a mis amigos José y Francisca, a mi mentor Dr. Benjamín Moreno, quienes me levantaron el ánimo en tiempos difíciles y me brindaron sus consejos para salir adelante.

Quiero expresar mi gratitud a la Dra. Graciela Román y al Dr. Manuel Aguilar, así como a mis asesores de tesis, el Dr. José Luis Quiroz y el Dr. Miguel Castro, por dedicar su tiempo para brindarme retroalimentación y consejos tanto a nivel personal como académico durante mi trayectoria en la maestría.

En especial, me gustaría agradecer al Dr. José Luis Quiroz por el apoyo moral que me ofreció en momentos en los que mi estado anímico afectó mi estilo de vida, salud y desempeño académico. Además, quiero destacar que él se ha convertido en un ejemplo de persona trabajadora, serena y humanista, a quien aspiro a ser algún día.

Índice general

Índice de figuras VI

Índice de tablas IX

1. Introducción **1**
1.1. Motivación 2
1.2. Objetivos 3
1.2.1. Objetivo general 3
1.2.2. Objetivos particulares 3
1.3. Organización de la tesis 3
1.3.1. Justificación 4
1.4. Metodología 5

2. Marco conceptual **7**
2.1. Programación en CUDA 7
2.2. Lenguajes visuales 9
2.3. Características de los lenguajes visuales 11

3. Trabajo relacionado **14**
3.1. CUDABlock 14
3.2. OpenCLGen 16
3.3. GPUBlocks 17

4. Gramática de grafos **19**

4.1. Introducción y terminología 20

4.1.1. Hiperarista 20

4.1.2. Hipergrafo 20

4.2. Reemplazo de Hiperarista 23

4.3. Gramática de grafos de reemplazo de hiperarista 25

4.4. Definición de un lenguaje de programación paralela mediante gramática de grafos 27

5. Lenguaje GLG **29**

5.1. Características de GLG 29

5.2. Estructuras visuales e íconos de GLG 30

5.2.1. Ícono de *Cabecera* 30

5.2.2. Ícono de *Constantes* 31

5.2.3. Ícono de *Variables de comunicación* 32

5.2.4. Íconos de *Procesamiento* 32

5.3. Gramática 35

5.3.1. Elementos de la grámatica GLG 35

5.3.2. Reglas de producción de la grámatica GLG 36

6. Ambiente GLG (GLG-E) **38**

6.1. Arquitectura de GLG 39

6.2. La interfaz 40

6.2.1. Traducción 42

6.2.2. Almacenamiento interno: Listas de de objetos 47

6.3. Motor de ejecución 52

6.3.1. La API-Rest 53

6.3.2. El entorno de CUDA remoto 54

7. Evaluación y resultados **57**

7.1. Comparación cualitativa 57

7.1.1. Evaluación respecto a programación textual 58

7.1.2. Evaluación respecto a lenguajes visuales 60

7.2. Comparación de desempeño 61
7.2.1. Casos de prueba 62
7.2.1.1. Diagramas de Voronoi 62
7.2.1.2. Área bajo la curva 62
7.2.2. Implementación de los problemas en GLG 63
7.2.2.1. Diagramas de Voronoi 64
7.2.2.2. Área bajo la curva 70
7.2.3. Evaluación de desempeño 74
7.2.3.1. Diagrama Voronoi 75
7.2.3.2. Área bajo la curva 75

8. Conclusiones y trabajo futuro **78**

Bibliografía **81**

Índice de figuras

2.1. Jerarquía de hilos en una tarjeta gráfica NVIDIA. 8
2.2. Ejemplo de organización de los hilos con base en el tamaño del *warp*. 8
2.3. Entorno de desarrollo de App Inventor. 10

3.1. Entorno de desarrollo de CUDABlock. 15
3.2. Entorno de desarrollo de OpenGLGen. 16
3.3. Entorno de desarrollo de GPUBlock. 17

4.1. Hiperarista de tipo 4. 20
4.2. Elementos del hipergrafo H. 22
4.3. Reemplazo de la hiperarista A en el grafo H por el hipergrafo R. . 23
4.4. Reemplazo de la hiperarista A en el hipergrafo H por el hipergrafo R. 24
4.5. Gramática de grafos de reemplazo de hiperarista. 26
4.6. Ejemplo de derivación en una gramática de reemplazo de hiperarista. 26

5.1. Ícono de *Cabecera* y *pop-up* correspondiente. 31
5.2. Ícono de *Constantes* y *pop-up* correspondiente. 31
5.3. Ícono de *Variables de comunicación* y *pop-up* correspondiente. . . 32
5.4. Ícono de procesamiento en la *CPU de entrada* y *pop-up* correspondiente. 33
5.5. Ícono de procesamiento en la *GPU* y *pop-up* correspondiente. . . 34
5.6. Ícono de procesamiento en la *CPU de salida* y *pop-up* correspondiente. 34
5.7. Ícono de *Salida* y *pop-up* correspondiente. 35

5.8. Reglas de producción de GLG. 36
5.9. Programa de ejemplo construido con la gramática de GLG. 37

6.1. Arquitectura de GLG-E. 40
6.2. Interfaz de la aplicación GLG-E. 41
6.3. Ejemplo de traducción de las propiedades del ícono de *Cabecera*. . 42
6.4. Ejemplo de traducción de las propiedades del ícono de *Constantes*. 43
6.5. Ejemplo de traducción para la definición de las variables de comunicación y su reserva de memoria en la CPU y GPU. 44
6.6. Ejemplo de traducción de las propiedades del ícono de *Variables de comunicación*. 45
6.7. Ejemplo de traducción de la definición, llamado al kernel y cálculo del número de bloques. 46
6.8. Ejemplo de la estructura del almacenamiento de datos de los íconos de GLG-E. 49
6.9. Ejemplo de la estructura de almacenamiento de las propiedades de las líneas. 50
6.10. Ejemplo del código construido a partir de las propiedades de los íconos de GLG-E. 51
6.11. Servicios de la API-Rest de GLG-E. 54

7.1. Diagrama de Voronoi en dos dimensiones. 62
7.2. Ejemplo de sumas de Riemman para la aproximación del área bajo la curva. 63
7.3. Ejemplo de asignación de filas a los hilos y cálculo de distancias euclidianas. 64
7.4. Selección de bibliotecas para el programa *diagramas de Voronoi*. . 65
7.5. Propiedades de los íconos de *Constantes* y *Variables de comunicación* para la creación del programa *diagramas de Voronoi*. 66
7.6. Asignación de generadores dentro de la matriz. 66
7.7. Llenado y selección de propiedades de los íconos de *CPU de entrada* y *salida* para la creación del programa *diagramas de Voronoi*. . . 67
7.8. Llenado de las propiedades del ícono de *GPU* para el programa *diagramas de Voronoi*. 68

7.9. Fragmento de código generado por GLG-E para el programa *diagramas de Voronoi*. 69
7.10. Salida del programa *diagramas de Voronoi*. 70
7.11. Ejemplo de la asignación de sub-intervalos a los hilos. 71
7.12. Propiedades de los íconos de *Cabecera* y *Constantes* para la creación del programa *área bajo la curva*. 72
7.13. Propiedades de los íconos de *Variables de comunicación* y *CPU de entrada* para la creación del programa *área bajo la curva*. 72
7.14. Propiedades de los íconos de *GPU* y *CPU de salida* para la creación del programa *área bajo la curva*. 73
7.15. Parcialidad del código generado y salida de la ejecución del programa *área bajo la curva*. 74
7.16. Tiempos del programa *diagramas de Voronoi*. 75
7.17. Tiempos del programa *área bajo la curva*. 76

Índice de tablas

3.1. Tabla comparativa de los lenguajes visuales. 18

7.1. Tabla comparativa de características de CUDA contra GLG. . . . 59
7.2. Tabla comparativa de los lenguajes visuales contra GLG. 60

Resumen

En las últimas décadas, el desarrollo de programas paralelos ha desempeñado un papel fundamental en la resolución eficiente de problemas científicos. Sin embargo, un buen programa paralelo debe tener en cuenta la arquitectura subyacente de ejecución.

En particular, el uso de tarjetas gráficas ha ganado popularidad debido a la gran cantidad de núcleos disponibles (miles de núcleos), lo que reduce significativamente el tiempo de ejecución en muchos casos. Sin embargo, el desarrollo de programas para tarjetas gráficas es complejo, ya que los programadores deben tener un conocimiento detallado de la arquitectura de la tarjeta para aprovecharla de manera eficiente.

En este trabajo se propone el lenguaje visual GLG (*Graphic Language for GPUs*) para construir programas en CUDA. Con GLG, los programas pueden crearse de forma sencilla y automatizada mediante la interconexión de iconos validados por una gramática. El conjunto de iconos presentado es configurable, ya que cada uno de ellos tiene diferentes parámetros (variables de comunicación, número de hilos, constantes, bibliotecas, etc.). Además de GLG, se propone un entorno de desarrollo web que utiliza una arquitectura Cliente-Servidor para enviar información al servidor y ejecutar los programas visuales.

Capítulo 1

Introducción

En la actualidad, existen una gran variedad de problemas que pueden resolverse mediante programas secuenciales. Sin embargo, hay otros que requieren un gran número de cálculos y cómputo intensivo, como el comportamiento del clima en un periodo de tiempo [ADM22], la evolución de las galaxias mediante simulaciones [ZB15], la mejora en la calidad de imagen de radiografías [ME09], el modelado de células cardíacas [NMRALG+20], entre otros.

Este tipo de problemas se resuelven mediante programas paralelos, los cuales se ejecutan en diferentes tipos de arquitecturas, procesadores multi-núcleo, clústeres y tarjetas gráficas. Estas últimas han ganado gran popularidad para la ejecución de programas paralelos, debido a su arquitectura que permite tener cientos o miles de núcleos trabajando al mismo tiempo, además de estar organizados en grupos de procesadores que comparten la misma memoria. Las tarjetas gráficas originalmente se diseñaron para el manejo de gráficos en videojuegos, pero desde hace tiempo también son utilizadas para el cómputo científico, la minería de datos y desarrollos de propósito general.

Sin embargo, programar de manera eficiente en tarjetas gráficas puede ser una tarea compleja, ya que el desarrollador debe tener un profundo conocimiento del problema a resolver, así como de la arquitectura de la tarjeta (número de multiprocesadores, número máximo de hilos, cantidad de memoria global, tamaño del *warp*, etc.), además de las herramientas y bibliotecas necesarias para la programación. Para aquellos usuarios con poca experiencia en programación,

considerar la arquitectura de la GPU para el desarrollo de programas implica un reto adicional, lo que dificulta en gran medida la escritura de programas [WM17].

Otro factor que aumenta la dificultad es la necesidad de paralelizar programas secuenciales. La paralelización no solo implica comprender los algoritmos y estructuras de datos, sino también optimizar y coordinar la ejecución simultánea de múltiples tareas. Las carreras de computación que introducen los conceptos de concurrencia y paralelismo tienen como principal objetivo enseñar el pensamiento paralelo [EM96, Mar15], pero debido al nivel de abstracción que se requiere puede llegar a ser complicado para los alumnos.

Una alternativa para reducir el problema de aprendizaje del pensamiento paralelo y del uso de tarjetas gráficas es el uso de lenguajes visuales [LTH15], [DMB16], [HLPT19], los cuales permiten ocultar ciertas instrucciones de comunicación y administración de procesos, así como el conocer a fondo la arquitectura de la tarjeta gráfica, haciendo que el uso de las tarjetas sea más sencillo para el programador [ESW17].

1.1. Motivación

En esta tesis, se propone el lenguaje visual GLG (*Graphic Language for GPUs*) para facilitar la construcción de programas en CUDA (*Compute Unified Device Architecture*) [Red22] de NVIDIA. Mediante íconos interconectados y validados por una gramática, GLG permite crear programas de forma sencilla. Cada ícono tiene diferentes propiedades, como variables de comunicación, número de hilos, constantes, bibliotecas, entre otros. Además de GLG, se propone un entorno de desarrollo que utiliza tecnologías web y una API-Rest para la ejecución de los programas visuales.

1.2. Objetivos

1.2.1. Objetivo general

Proponer e implementar un lenguaje visual y su entorno de desarrollo para facilitar la creación de programas paralelos en CUDA.

1.2.2. Objetivos particulares

1. Identificar los elementos de procesamiento, comunicación y sincronización básicos para el desarrollo de un programa en la plataforma CUDA.
2. Definir un lenguaje visual básico junto con su gramática para identificar las operaciones fundamentales en la plataforma CUDA.
3. Definir un conjunto de íconos para identificar las diferentes operaciones al construir programas en CUDA.
4. Implementar una API-Rest (API construida con base en el protocolo HTTP) para la ejecución de programas en CUDA a través de un navegador web.
5. Implementar el nuevo lenguaje y su entorno de desarrollo mediante tecnologías web.
6. Comparar el lenguaje propuesto con otros lenguajes visuales encontrados en la literatura.
7. Validar el rendimiento del lenguaje propuesto contra la programación textual.

1.3. Organización de la tesis

La presente tesis está estructurada de la siguiente forma:

- En este primer Capítulo se presenta el planteamiento general de la tesis.

- El Capítulo 2 proporciona un marco conceptual para comprender los fundamentos de la programación en CUDA, las características de los lenguajes visuales y sus ventajas contra los lenguajes textuales.

- En el Capítulo 3 se presenta el estado del arte de lenguajes visuales encontrados en la literatura, describiendo sus características y modelos de programación. Este capítulo permite conocer las ventajas y desventajas de los ambientes y lenguajes visuales existentes.

- En el Capítulo 4 se expone la gramática de grafos, mediante el reemplazo de hiperaristas. Esta parte introduce los conceptos requeridos para entender la gramática propuesta para el lenguaje GLG.

- El Capítulo 5 presenta el lenguaje propuesto GLG. Se describen y clasifican cada unos de sus elementos gráficos. También se muestra la gramática de GLG y se describe la función de cada una de sus reglas de producción.

- El Capítulo 6 describe la arquitectura del entorno de desarrollo de GLG. Se describe la función de cada módulo en GLG a fin de traducir un programa visual de GLG a CUDA para ejecutarlo en una tarjeta gráfica NVIDIA.

- En el Capítulo 7, se realiza una evaluación cualitativa de GLG-E en comparación con lenguajes textuales y visuales. Además, se presentan casos de prueba desarrollados con GLG-E para ilustrar su aplicabilidad y se realiza una medición de su eficiencia y rendimiento.

- El Capítulo 8 cierra con las conclusiones y las sugerencias para el trabajo futuro.

- La bibliografía de este trabajo se encuentra al final del documento.

1.3.1. Justificación

Para aprovechar el potencial de las tarjetas gráficas NVIDIA en el desarrollo de programas paralelos presenta desafíos significativos en términos de complejidad y optimización. La programación en CUDA, aunque poderosa, puede ser compleja

y propensa a errores, especialmente para aquellos que carecen de un profundo conocimiento de la arquitectura de las GPUs.

Por esto surge la necesidad de crear herramientas que faciliten la creación de programas paralelos de manera más accesible, eficiente y visual. Este trabajo de investigación se centra en esta necesidad mediante la creación y evaluación de GLG, un lenguaje visual diseñado específicamente para la programación en tarjetas gráficas NVIDIA.

La importancia de este trabajo radica en la posibilidad de facilitar el desarrollo de aplicaciones paralelas, permitiendo que un conjunto más amplio de programadores pueda aprovechar los beneficios del cómputo en GPU sin enfrentar la curva de aprendizaje pronunciada que suele estar asociada con la programación textual en CUDA. Además, GLG no solo simplifica la creación de código, sino que también busca mantener un alto nivel de control y eficiencia, permitiendo que los desarrolladores puedan ajustar manualmente el código generado según sea necesario.

Tiene como objetivo ser una herramienta par facilitar la programación a investigadores, profesores y estudiantes para aplicarlo en cursos de programación. Esto permite al programador enfocarse en entender la sintaxis de CUDA y no gastar tiempo en la corrección de errores de compilación.

Mediante la evaluación cualitativa y cuantitativa de GLG en comparación con enfoques tradicionales y otras herramientas visuales, este trabajo busca demostrar su efectividad en términos de expresividad, aplicabilidad, eficiencia y facilidad de uso. Al ofrecer una alternativa más amigable para el desarrollo de programas paralelos en tarjetas gráficas, este enfoque podría tener un impacto significativo en la forma en que se abordan problemas computacionalmente intensivos en diversas áreas como la ciencia, la ingeniería y la investigación.

1.4. Metodología

Para asegurar el correcto desarrollo del proceso de investigación del proyecto de Maestría en Ciencias y Tecnologías de la Información, se toma como base la experiencia adquirida en la definición e implementación del lenguaje VPPL y

VPPE, realizada por los asesores de tesis [Fab19]. A continuación, se enumeran las actividades a realizar para concluir el proyecto en tiempo y forma:

- Revisión del estado del arte de los lenguajes visuales para el desarrollo sobre tarjetas gráficas.
- Identificación de los elementos básicos para el desarrollo de programas en CUDA, entre los que podemos mencionar: obtener el número máximo de bloques e hilos a crear, la transferencia de datos de memoria RAM a memoria global de la GPU y viceversa, la invocación de un kernel en la GPU, entre otros.
- Definición de un lenguaje visual, considerando los elementos identificados en el punto anterior, y formalizarlo mediante el uso de una gramática junto con sus reglas de producción.
- Definición de las propiedades básicas de cada elemento del lenguaje visual, por ejemplo, el código fuente del programa en CUDA, los parámetros del programa, el número de bloques o hilos, etc.
- Definición del esquema de traducción del lenguaje visual propuesto a código de un programa en CUDA.
- Definición de una API-Rest que permita la ejecución de programas mediante CUDA. El desarrollo de la API-Rest se puede facilitar con Node.js, el cual permite la ejecución de programas escritos en Javascript sobre un servidor.
- Implementación del entorno de desarrollo mediante Angular (*framework* utilizado para el desarrollo web) el cual permite crear y mantener aplicaciones web de una sola página(*single page application*).
- Conexión de la API-Rest con el entorno de desarrollo a fin de ejecutar programas visuales del lenguaje propuesto mediante un navegador web.
- Comparar los resultados con el desarrollo textual tradicional.
- Reportar los resultados en la Idónea Comunicación de Resultados.

Capítulo 2

Marco conceptual

En este capítulo, se realiza una comparación de los lenguajes y entornos visuales más representativos utilizados para el desarrollo de programas paralelos en tarjetas gráficas. En primer lugar, se presentan brevemente las desventajas de utilizar la programación textual tradicional en tarjetas gráficas, seguido de las ventajas de realizar la programación con lenguajes visuales, especialmente aquellos enfocados en flujos de trabajo (serie de tareas). Por último, se examinan los lenguajes más representativos en términos de su expresividad, aplicabilidad, nivel de interacción y portabilidad.

2.1. Programación en CUDA

CUDA es una plataforma de programación paralela desarrollada por NVIDIA, la cual permite aprovechar el potencial de procesamiento de las tarjetas gráficas para realizar cálculos intensivos de manera más eficiente en comparación de una CPU convencional.

CUDA implementa un modelo de ejecución llamado SIMT (*Single Instruction Multiple Threads*) considerando que las tarjetas gráficas están conformados por varios multiprocesadores[1]. Tradicionalmente, el desarrollo de programas paralelos en CUDA se lleva a cabo utilizando lenguajes o bibliotecas de programación

[1] Cada multiprocesador esta conformado por varios núcleos (*cores*)

textuales, y en particular se basa en tres abstracciones básicas: la jerarquía de grupos de hilos, los tipos de memoria y las barreras de sincronización. Los hilos que se ejecutan en la tarjeta NVIDIA (*device*) están estructurados en una jerarquía que incluye mallas (*grids*) compuestas por bloques de una, dos o tres dimensiones, y los bloques están formados por grupos de hilos igualmente de una, dos o tres dimensiones. Por último, el hilo es la unidad elemental de ejecución. Esta estructura se puede visualizar en la Figura 2.1.

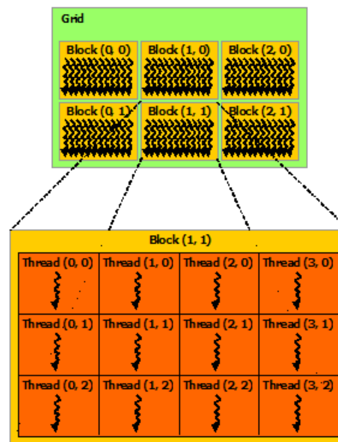


Figura 2.1: Jerarquía de hilos en una tarjeta gráfica NVIDIA.

Durante la ejecución de un programa en CUDA, los multiprocesadores pueden recibir uno o más bloques. Cuando un bloque se asigna a un multiprocesador, debe ser completamente ejecutado por el multiprocesador. Cada bloque de hilos activos es dividido en grupos llamados **warps** (Figura 2.2), los cuales son ejecutados en el multiprocesador de forma SIMT (*Single Instruction Multiple Threads*). Es decir, todos los hilos dentro del *warp* ejecutan la misma instrucción cada vez.

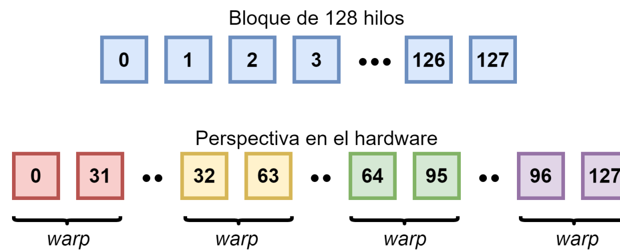


Figura 2.2: Ejemplo de organización de los hilos con base en el tamaño del *warp*.

Dado que las tarjetas gráficas está instaladas en una computadora, se tiene un modelo híbrido para la programación en el que se combina el código del programa principal que se ejecuta en la CPU (*host*) y las llamadas a funciones que se ejecutan paralelamente por varios hilos en la GPU (*device*), conocidas como funciones kernel. En cada llamada a un kernel, se deben especificar la cantidad de hilos por bloque con los que se ejecutará la función kernel en la GPU.

Los hilos tienen acceso a diferentes espacios de memoria: global, local, compartida, textura y de registros. Cada hilo tiene un espacio de memoria local privado. A su vez, cada bloque de hilos posee memoria compartida, visible solo para todos los hilos del bloque y con la misma duración de vida que el bloque. Todos los hilos ejecutados por una función kernel tienen acceso a la misma memoria global. Los espacios de memoria global, constante y de textura son persistentes en diferentes activaciones de funciones kernels en la misma aplicación. El trabajo de estos hilos puede ser sincronizado mediante directivas de bloqueo, lo cual posibilita la coordinación entre ellos. Sin embargo, los hilos agrupados en diferentes bloques no pueden comunicarse entre sí.

Programar en CUDA puede ser considerado como un desafío por el conocimiento sólido que se debe tener de programación paralela y de la arquitectura de la tarjeta. La complejidad de programar en CUDA depende de varios factores, como la complejidad del algoritmo que se desea implementar, el tamaño y tipo de los datos, y la optimización que se busca lograr.

Una alternativa para programar utilizando estas tecnologías de cómputo paralelo son los lenguajes visuales que permiten construir el código de bajo nivel del propio lenguaje mediante figuras o íconos interconectados entre sí.

2.2. Lenguajes visuales

En el ámbito de la programación, un lenguaje visual es un tipo de lenguaje de programación que se basa en la representación gráfica de conceptos y elementos, a diferencia de los lenguajes de programación textuales que se centran en la escritura de código en forma de texto. Los lenguajes de programación visuales utilizan íconos para expresar la semántica [ESW17]. Mediante elementos gráficos,

el usuario puede construir programas en lugar de diseñarlos de manera textual.

En lugar de expresar instrucciones y algoritmos mediante líneas de código escritas, los lenguajes visuales permiten a los desarrolladores construir programas manipulando elementos gráficos en una interfaz gráfica. Esta representación visual puede ser beneficiosa para comprender y comunicar conceptos complejos, especialmente en el ámbito de la programación paralela, donde la coordinación de múltiples tareas y flujos de datos puede ser complicada.

Los lenguajes visuales tienen la ventaja de que pueden hacer que la programación sea más accesible para personas que pueden no tener experiencia en programación textual, ya que las representaciones visuales pueden ser más intuitivas y fáciles de comprender. Sin embargo, la efectividad de un lenguaje visual depende en gran medida de su capacidad para representar claramente las estructuras y relaciones del programa, así como de su capacidad para generar código ejecutable eficiente en segundo plano.

Algunos lenguajes suelen ser utilizados en ámbitos educativos para introducir a los jóvenes en el mundo de la programación. Un ejemplo de ello es App Inventor [MIT09], que permite crear aplicaciones móviles simples para Android arrastrando módulos y creando estructuras de control cíclicas, condicionales o iterativas, así como variables y constantes, entre otros elementos. Estos módulos están diseñados para encajar como las piezas de un rompecabezas, como se muestra en la Figura 2.3.

Figura 2.3: Entorno de desarrollo de App Inventor.

Sin embargo, existen otros tipos de lenguajes visuales enfocados en el de-

sarrollo de programas paralelos, los cuales se pueden clasificar en aquellos que utilizan arquitecturas con CPU (unidad central de procesamiento) y los que utilizan arquitecturas con GPU (unidad de procesamiento gráfico). En este trabajo nos enfocamos en los lenguajes que permiten el desarrollo de programas en tarjetas gráficas, ya que un diagrama visual permite plasmar de manera más efectiva la idea del programador en cuanto al flujo de trabajo, a diferencia de la versión textual.

2.3. Características de los lenguajes visuales

Como se mencionó en la sección anterior, un flujo de trabajo es una secuencia de tareas, las cuales se pueden visualizar de forma paralela mediante un grafo que organiza el flujo de datos. Por lo tanto, es más fácil plasmar la idea de forma visual, ya que se tiene una percepción más clara de la solución paralela del problema al utilizar un diagrama. Para lograr este nivel de abstracción, el lenguaje visual debe cumplir con diferentes tipos de íconos, comportamientos y características que, de acuerdo con [Doz01], permiten evaluar cada sistema en términos de su expresividad, aplicabilidad, nivel de interacción y portabilidad.

- **Expresividad:** La expresividad se refiere a la cantidad de elementos visuales (íconos) presentes en la aplicación, así como a las propiedades o funciones que facilitan el desarrollo de algoritmos por parte de los programadores. Entre estas propiedades se incluyen: tareas secuenciales, estructuras de selección, estructuras de repetición, operaciones de entrada y salida, patrones de procesamiento especial y comunicación. Es importante destacar que en el lenguaje CUDA, la comunicación de datos entre la CPU y la GPU puede resultar complicada para los desarrolladores, ya que deben reservar memoria en ambos dispositivos y luego realizar la copia mediante palabras reservadas (por ejemplo, *cudaMemcpy*), además de especificar el nombre de la variable tanto en la CPU como en la GPU.

- **Aplicabilidad:** El rango de programas que pueden ser creados con un lenguaje visual y que abarca una amplia variedad de problemas a resolver es

a lo que se define como aplicabilidad. Todos los entornos de programación permiten construir programas utilizando patrones con el objetivo de incrementar esta característica.

La definición de una gramática bien estructurada puede determinar el rango de programas que es posible crear. En el entorno de desarrollo propuesto por [Fab19], se utilizan íconos de repetición y bifurcación que emplean los patrones SPMD (*Single Program Multiple Data*) y MPMD (*Multiple Program Multiple Data*), respectivamente. Aunque todos los entornos cuentan al menos con un patrón de procesamiento paralelo, el número y tipo de estos entornos suelen ser bastante limitados. Además, no se consideran patrones que permitan trabajar con programas que utilicen conjuntos de datos dinámicos, como el patrón Maestro-Esclavo.

- **Nivel de interacción:** Se refiere al grado de facilidad para acceder y navegar por la aplicación de forma intuitiva, sencilla y rápida. Esto se logra mediante una buena interfaz de usuario (*User Interface*), que comprende los elementos visuales que permiten al usuario interactuar adecuadamente con la aplicación, así como una buena experiencia de usuario (*User Experience*), que se refiere a las emociones experimentadas por el usuario antes, durante y después de utilizar la aplicación.

  Entre los aspectos fundamentales a considerar en el nivel de interacción se encuentran: la interfaz de arrastrar y soltar (*drag-and-drop*), la encapsulación de íconos, el reordenamiento de íconos, el acercamiento y alejamiento del flujo de trabajo (*zoom*), y el acceso a la nube.

- **Portabilidad:** Se refiere a la capacidad del lenguaje visual para ser ejecutado en diferentes plataformas. La limitada popularidad y distribución de los entornos visuales se debe, en parte, a esta falta de portabilidad [Doz01]. Uno de los aspectos favorables en un lenguaje visual es la independencia entre el apartado visual (*front-end*) y la lógica de negocio (*back-end*). Esto permite utilizar la aplicación en cualquier plataforma y establecer una conexión a través de un intermediario (API-Rest) con el módulo que se encarga de toda la lógica de negocio, que actualmente se implementa en la nube.

Resumen

En este capítulo se realizó un estudio de lenguajes y entornos visuales, considerando diversas características importantes para un lenguaje visual según lo propuesto en [Doz01]: expresividad, aplicabilidad, nivel de interacción y portabilidad.

Capítulo 3

Trabajo relacionado

Entre los lenguajes visuales que han sido propuestos para desarrollo sobre tarjetas gráficas se encuentran: CUDABlock [LTH15], OpenCLGen [DMB16], GPU-Blocks [HLPT19]. A continuación se presentan las características de cada uno de ellos, así como sus ventajas y desventajas.

3.1. CUDABlock

CUDABlock [LTH15] es un lenguaje visual diseñado para el desarrollo de programas en CUDA. Fue creado utilizando la biblioteca *OpenBlocks* de Java [Roq08], la cual facilita la creación de íconos y la definición de su comportamiento. Este lenguaje visual permite construir la estructura de un programa de CUDA en forma de puzzle mediante su interfaz (Figura 3.1). CUDABlock consta de 5 tipos de íconos: bloques de control, bloques de prueba, bloques matemáticos, bloques de entrada y salida, y bloques de variables y constantes. Visualmente, cada uno de estos íconos tiene un conjunto de propiedades específicas que corresponden con su funcionalidad, lo que facilita la identificación y el ensamblaje adecuado. La ventaja de CUDABlock radica en su enfoque de optimización en el procesamiento de datos, lo que permite obtener programas más rápidos y eficientes.

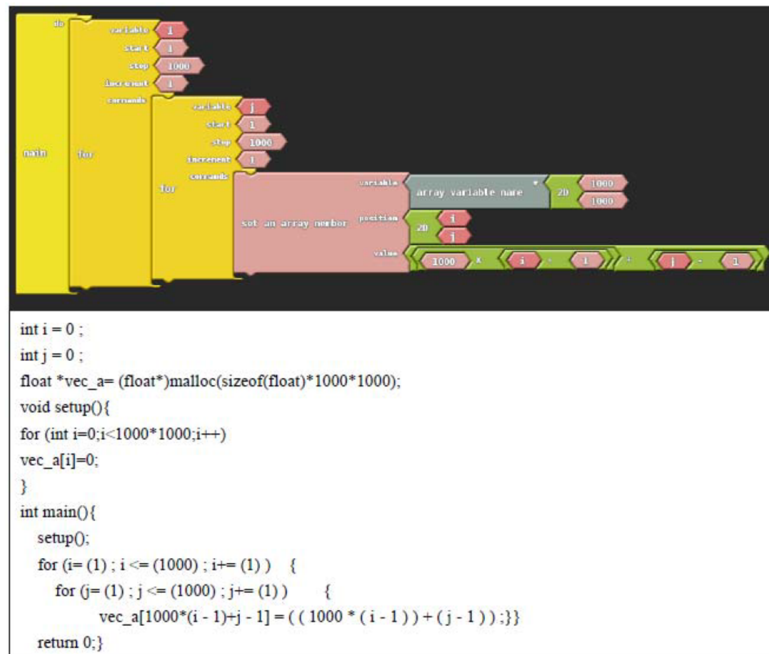


Figura 3.1: Entorno de desarrollo de CUDABlock.

Entre los programas que han sido desarrollados utilizando este lenguaje para probar su eficiencia se incluyen operaciones como la suma, resta, multiplicación, multiplicación escalar y convoluciones de arreglos de dimensión 1 y 2. Sin embargo, una desventaja importante de CUDABlock es que utiliza múltiples íconos (más de 20) que corresponden a operaciones específicas, lo que requiere que el usuario tenga un sólido conocimiento de CUDA y C para utilizar el lenguaje de manera óptima. Aunque la variedad de íconos aumenta la expresividad del lenguaje, también disminuye el nivel de interacción, ya que el usuario necesita invertir más tiempo para conocer las propiedades de cada ícono, además de comprender el código CUDA y C subyacente.

En términos de aplicabilidad, es posible construir una gran variedad de programas utilizando CUDABlock en lugar de escribirlos de forma tradicional, gracias a la expresividad del lenguaje. Sin embargo, es importante tener en cuenta que CUDABlock está fuertemente ligado al dispositivo en el que se ejecuta, lo que limita su portabilidad y hace que el acceso a la nube sea prácticamente nulo.

3.2. OpenCLGen

En el artículo [DMB16], se presenta OpenCLGen, una aplicación web (Figura 3.2) que genera código OpenCL para un kernel a la vez. Esta herramienta cuenta con un modelo de ejecución, manejo de errores y campos de entrada donde se pueden ingresar los datos necesarios. La arquitectura del software requiere que el programador identifique la sección de su programa secuencial que se desea paralelizar, y luego define su comportamiento.

Una ventaja de la arquitectura de OpenCLGen es que está modularizada en diferentes partes: los datos de entrada, que incluyen el nombre del kernel, el número y tipo de datos, y el dispositivo de ejecución (CPU o GPU); el dominio lógico, encargado de construir el código basándose en la información proporcionada por el usuario; y los datos de salida, que incluyen el código que se ejecuta en el *host* y en el *device*, así como los directorios de cabecera necesarios.

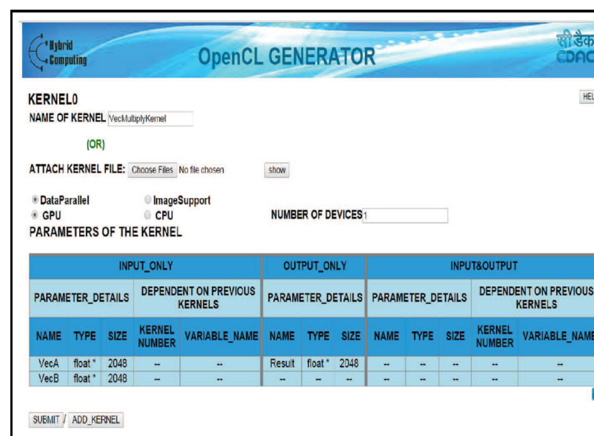


Figura 3.2: Entorno de desarrollo de OpenGLGen.

El código generado por OpenCLGen es ejecutado en un servidor utilizando una API-Rest, a la cual se envían peticiones mediante el protocolo HTTP, lo cual favorece la portabilidad del lenguaje, ya que no depende exclusivamente de la arquitectura del dispositivo en el que se ejecute la aplicación web. Sin embargo, la interfaz de usuario se compone de ventanas con campos que el usuario debe llenar para generar el código por partes, lo que puede resultar confuso para el usuario. Además, la herramienta carece de elementos *drag-and-drop* (arrastrar y

soltar), lo que limita su expresividad y, como resultado, su nivel de interacción es bajo.

3.3. GPUBlocks

GPUBlocks [HLPT19] es un trabajo que extiende de [LTH15], ya que además de fungir como entorno para CUDA, también lo es para OpenCL. Esta herramienta visual facilita la construcción de códigos para arquitecturas *manycore* (tarjetas gráficas). La interfaz (Figura 3.3) desarrollada con *OpenBlocks* facilita la creación de bloques arrastrables, sin embargo en algunos se deben utilizar sentencias específicas de CUDA u OpenCL, tal como ocurre con CUDABlock, por lo que el usuario debe tener conocimientos avanzados en el lenguaje.

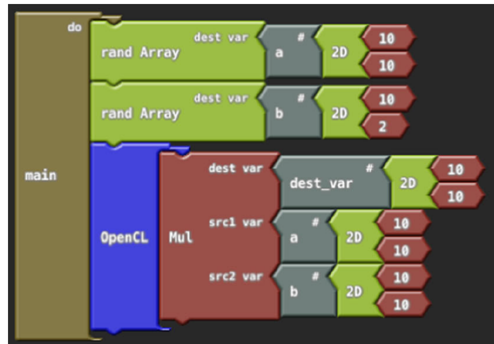


Figura 3.3: Entorno de desarrollo de GPUBlock.

Una comparación de estos trabajos se visualiza en la Tabla 3.1, donde se muestra cómo se desempeñan cada una de las herramientas en relación con las características mencionadas en la Sección 2.3. La cantidad de asteriscos asignados en la tabla indican la puntuación correspondiente para cada herramienta, donde una mayor cantidad de asteriscos representa un mejor desempeño en la característica evaluada.

Característica	CUDABlock [LTH15]	OpenCLGen [DMB16]	GPUBlocks [HLPT19]
Expresividad	****	*	****
Aplicabilidad	***	**	***
Nivel de Interacción	****	**	****
Portabilidad	***	*****	***

Tabla 3.1: Tabla comparativa de los lenguajes visuales.

En general, se pueden extraer las siguientes conclusiones de la tabla anterior:

- Existe una relación directa entre la expresividad de un lenguaje y su aplicabilidad. Cuanto mayor sea el número de íconos disponibles, más control se tiene sobre la tarjeta gráfica.

- El nivel de interacción es un aspecto crucial, ya que un menor número de clics y una interfaz intuitiva facilitan la construcción de programas y mejoran la experiencia del usuario.

- La portabilidad es un factor importante, ya que permite utilizar las herramientas en diferentes dispositivos sin depender de su arquitectura. El uso de servidores en la nube y una API-Rest es una opción favorable en este sentido.

Para finalizar, resultaría ideal para la programación en tarjetas gráficas un entorno de desarrollo web con la funcionalidad de arrastrar y soltar íconos, ventanas emergentes, capacidad de procesamiento en la nube para ejecutar CUDA de forma remota y una API-Rest para el intercambio de datos.

Resumen

En este capítulo se hizo énfasis en los lenguajes visuales que utilizan flujos de trabajo para representar la concurrencia y el flujo de datos. Se compararon tres herramientas con base en estas características: CUDABlock, OpenCLGen y GPUBlocks.

Capítulo 4

Gramática de grafos

La gramática de grafos es una herramienta utilizada en el campo de la teoría de lenguajes formales y la teoría de grafos. Se utiliza para definir la estructura y las reglas sintácticas de un lenguaje visual basado en grafos en lugar de utilizar reglas gramaticales tradicionales como en las gramáticas formales [DKH17]. La gramática de grafos se basa en nodos y arcos para representar los elementos y las relaciones entre ellos. Los nodos representan los elementos individuales, como símbolos *terminales* o *no terminales*, mientras que los arcos representan las conexiones o relaciones entre los nodos.

La gramática de grafos define cómo se pueden combinar los nodos y los arcos para formar estructuras válidas dentro del lenguaje visual. Estas reglas sintácticas determinan la forma en que se pueden conectar los nodos y las restricciones en las relaciones entre ellos. Al definir una gramática de grafos, se establecen las reglas sintácticas para construir diagramas o programas visuales de acuerdo con la semántica del lenguaje específico. Esto permite validar la conexión sintáctica de los programas creados en el entorno visual y proporciona una guía para la construcción correcta de las estructuras visuales.

4.1. Introducción y terminología

4.1.1. Hiperarista

Una hiperarista es un concepto utilizado para representar una relación entre más de dos nodos en un grafo. Mientras que una arista convencional conecta únicamente dos nodos, una hiperarista permite conectar tres o más nodos simultáneamente mediante arcos. De esta manera se proporciona una forma de modelar conexiones más ricas y flexibles que van más allá de las conexiones binarias tradicionales de las aristas.

El tipo de hiperarista es equivalente al número de sus arcos. En el ejemplo de la Figura 4.1 se muestra una hiperarista con 4 arcos, por lo tanto es de tipo 4.

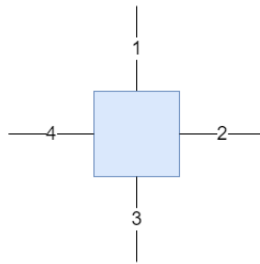


Figura 4.1: Hiperarista de tipo 4.

4.1.2. Hipergrafo

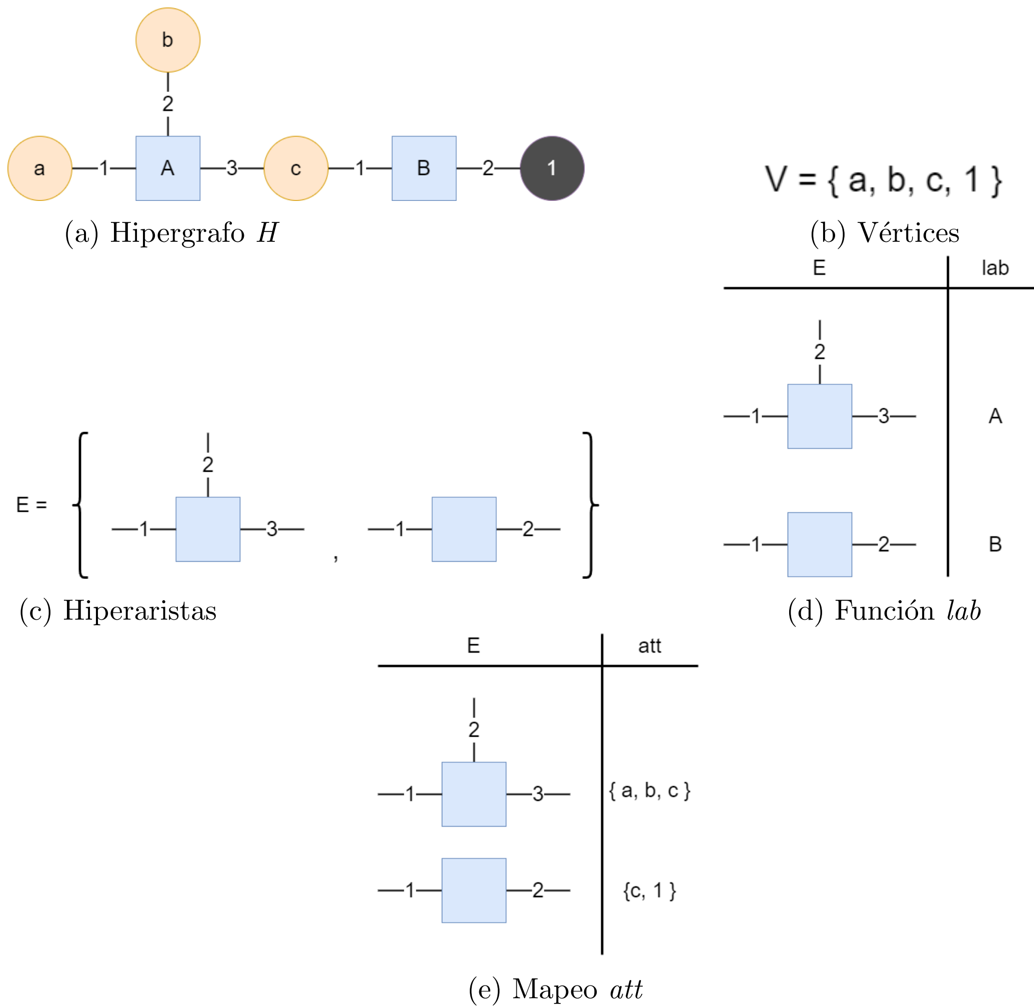
Un hipergrafo permite representar relaciones entre conjuntos de nodos a diferencia de los grafos convencionales, donde las aristas conectan únicamente dos nodos. Formalmente, un hipergrafo se define como una 4-tupla (*V, E, lab, att*), siendo C un conjunto de etiquetas y H un hipergrafo sobre C, entonces:

- V (*vertices*) es un conjunto de nodos o vértices
- E (*edge*) es un conjunto de hiperaristas tal que $V \cap E = \emptyset$
- *lab* es una función que etiqueta cada hiperarista, $lab : E \to C$

- *att* es un mapeo que contiene el conjunto de nodos que están asociados (conectados) a una hiperarista $e \in E$ que satisface $tipo(e) = |att(e)|$
  $att : E \to V^*$

Para ejemplificar esta definición se puede observar el hipergrafo H de la Figura 4.2a, el cual es de tipo 1, ya que este valor corresponde al número de nodos externos que posee (un círculo negro). Los nodos de este se pueden clasificar en dos tipos: externos, que permiten conectar a dos hipergrafos, y los internos, que son los nodos de un grafo convencional. A continuación, se explican cada uno de sus componentes:

- Los vértices del hipergrafo H están representados en la Figura 4.2b.
- La Figura 4.2c muestra el conjunto E de las hiperaristas junto con sus arcos.
- La función de etiquetado *lab* (Figura 4.2d) sirve para nombrar a cada hiperarista asignando dentro de su cuadrado una etiqueta; en el documento nos referiremos a una hiperarista nombrando a su etiqueta asociada.
- El mapeo *att* de cada hiperarista del hipergrafo H (Figura 4.2e) especifica los nodos que están conectados con los arcos de cada hiperarista.



(a) Hipergrafo H

(b) Vértices

(c) Hiperaristas

(d) Función lab

(e) Mapeo att

Figura 4.2: Elementos del hipergrafo H.

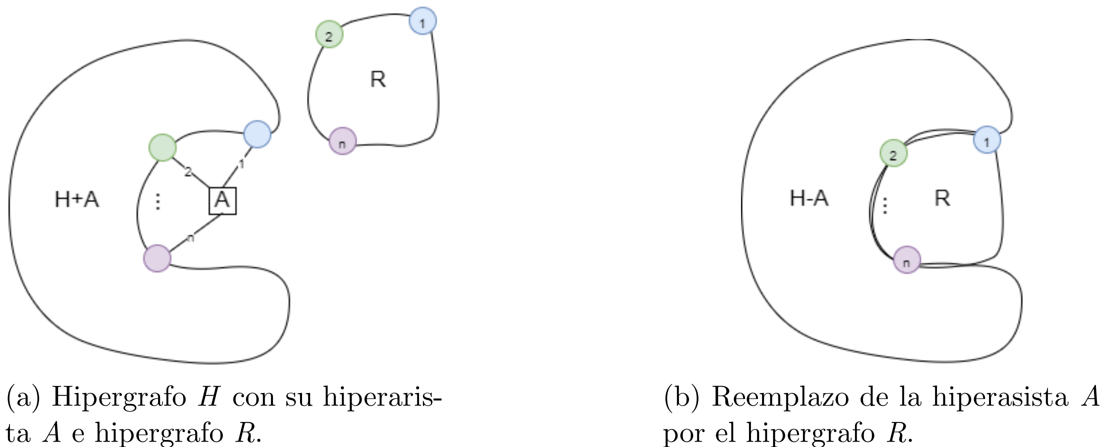
4.2. Reemplazo de Hiperarista

El reemplazo de hiperarista es una operación fundamental en la teoría de grafos y hipergrafos. Consiste en reemplazar una hiperarista existente en un hipergrafo por otra hiperarista o conjunto de hiperaristas.

Es importante destacar que el reemplazo de hiperarista puede tener implicaciones en la conectividad y propiedades estructurales del hipergrafo, y su aplicación debe realizarse cuidadosamente teniendo en cuenta las restricciones y reglas definidas para el sistema o modelo en particular en el que se esté trabajando. Para ilustrar el reemplazo de hiperaristas en hipergrafos se presentan dos ejemplos.

Ejemplo 1

En la Figura 4.3 se muestra el hipergrafo H, que tiene una hiperarista A con n arcos. R es otro hipergrafo con n nodos (Figura 4.3a). Después, A es reemplazada por el hipergrafo R manteniendo el número de nodos de H y desapareciendo los nodos externos de R (ver Figura 4.3b).



(a) Hipergrafo H con su hiperarista A e hipergrafo R.

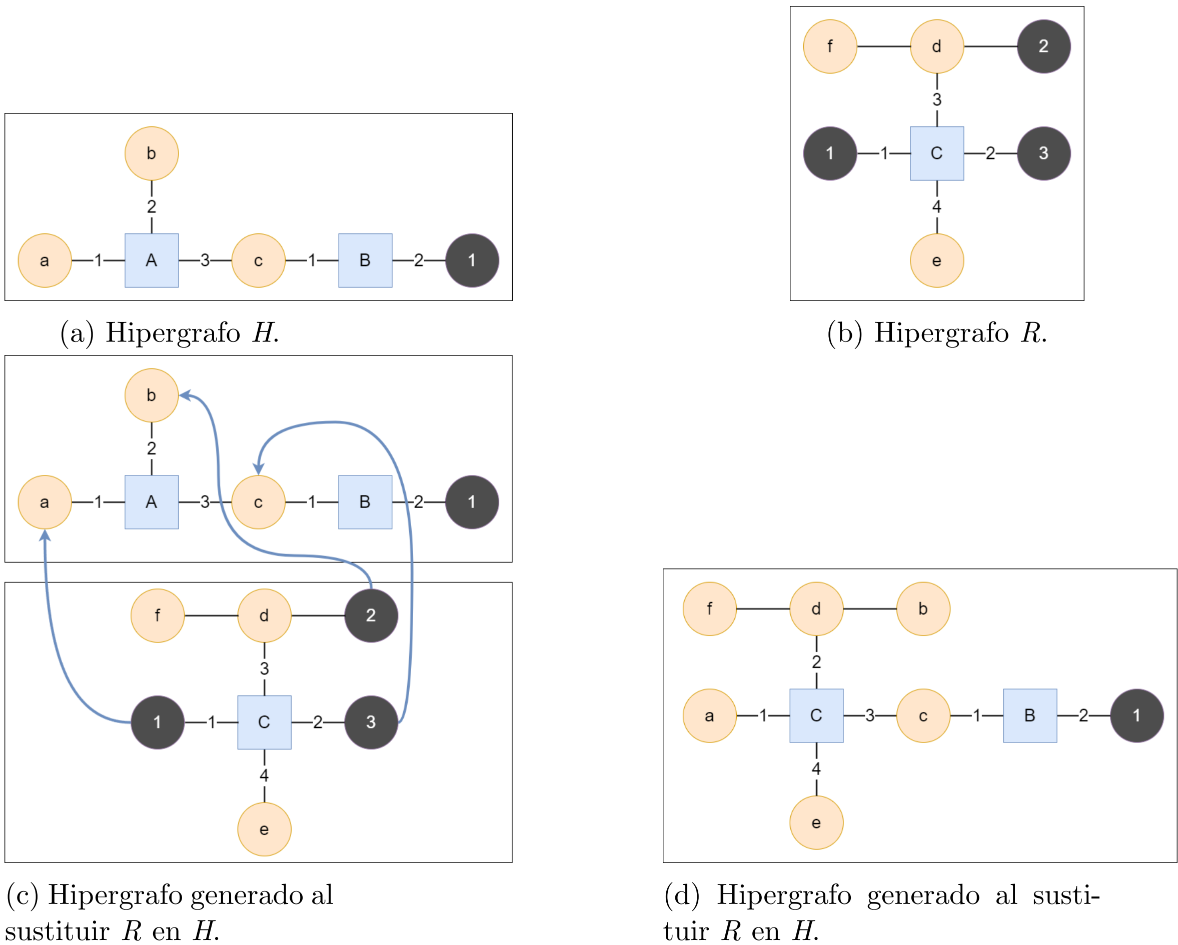
(b) Reemplazo de la hiperasista A por el hipergrafo R.

Figura 4.3: Reemplazo de la hiperarista A en el grafo H por el hipergrafo R.

Ejemplo 2

En la Figura 4.4a se tiene un hipergrafo H que incluye una hiperarista A, que a su vez tiene conectados los nodos $\{a, b, c\}$, por lo tanto es de tipo 3. Además, se tiene un hipergrafo R con 3 nodos externos identificados con el color negro (tipo(R) = 3), ver Figura 4.4b. El hipergrafo R puede reemplazar a la hiperarista A en H como se muestra en la Figura 4.4c de la siguiente forma:

1. La hiperarista A es reemplazada por la hiperarista C
2. El nodo externo *1* de R se conecta con el nodo a de H
3. El nodo externo *2* de R se conecta con el nodo b de H
4. El nodo externo *3* de R se conecta con el nodo c de H



(a) Hipergrafo H.

(b) Hipergrafo R.

(c) Hipergrafo generado al sustituir R en H.

(d) Hipergrafo generado al sustituir R en H.

Figura 4.4: Reemplazo de la hiperarista A en el hipergrafo H por el hipergrafo R.

Finalmente se obtiene un nuevo hipergrafo del reemplazo de R en H como se observa en la Figura 4.4d.

En la siguiente sección se profundizará en el desarrollo de las reglas y restricciones para describir la transformación de hiperaristas en un hipergrafo. Estas

reglas permitirán definir la sintaxis y la semántica del lenguaje utilizado para manipular y transformar las estructuras hipergráficas.

4.3. Gramática de grafos de reemplazo de hiperarista

Es un formalismo utilizado para describir la estructura y generación de hipergrafos. Se definen reglas de reemplazo que especifican cómo se puede reescribir una hiperarista en función de su contexto y las producciones permitidas. Cada regla de reemplazo consta de un lado izquierdo (hiperarista a ser reemplazada) y un lado derecho (la nueva configuración de hiperaristas después de la reescritura).

Esta gramática es muy similar a las gramáticas textuales, se tienen un conjunto de *no terminales* (representados por las etiquetas de las hiperaristas), *terminales* (representados por los nodos) y un conjunto de reglas de producción que permite identificar que hipergrafos pueden reemplazar una determinada hiperarista. La definición formal de una gramática de grafos de reemplazo de hiperaristas es la siguiente:

Sea C un conjunto de etiquetas, una grámatica de grafos de reemplazo de hiperaristas (GGRH) sobre C es una 4-tupla G $= (N, \Sigma, R, S)$, donde:

- $N, \Sigma \subseteq C$, son conjuntos de etiquetas de *no terminales* (hiperaristas) y *terminales* (nodos internos) finitos y disjuntos ($N \cap \Sigma = \emptyset$)
- $S \in N$ es el *no terminal* inicial.
- R es un conjunto de reglas de la forma $A ::= H$ con $A \in N$ y H es un hipergrafo tal que, $tipo(A) = tipo(H)$.

La Figura 4.5 muestra un ejemplo de GGRH con cuatro reglas de producción (hipergrafo 1, 2, 3 y 4) y siendo S la hiperarista inicial.

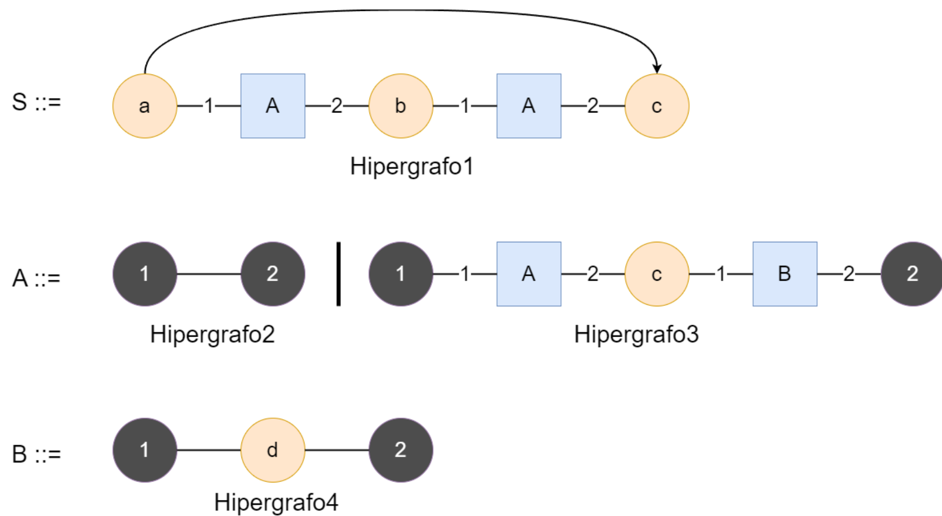


Figura 4.5: Gramática de grafos de reemplazo de hiperarista.

Un ejemplo de una derivación usando la gramática anterior se muestra en la Figura 4.6. El hipergrafo inicial se muestra de color rojo. Los hipergrafos de lado izquierdo señalan con una región morada a los nodos o hiperaristas que se le aplican las reglas correspondientes. De lado derecho se muestra el resultado de haber aplicado la regla. En color verde se muestra el hipergrafo final.

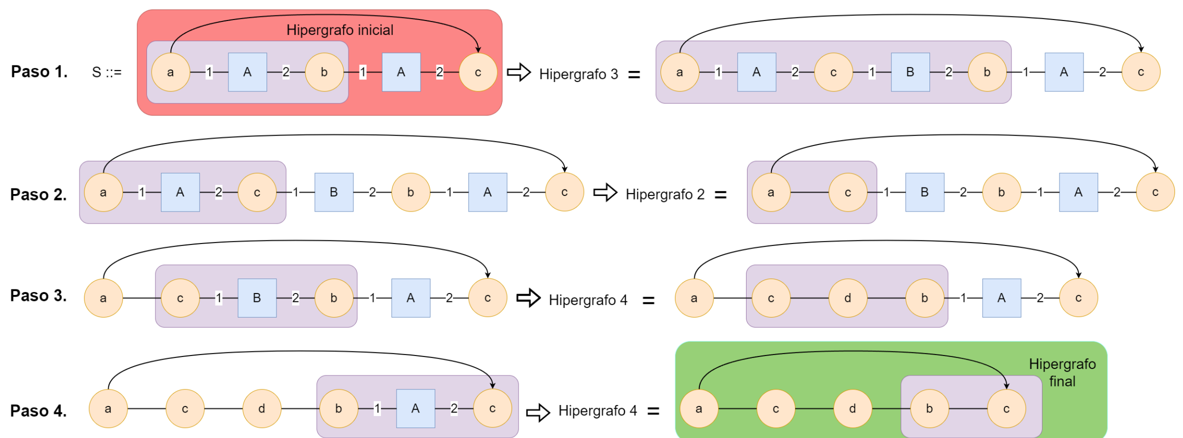


Figura 4.6: Ejemplo de derivación en una gramática de reemplazo de hiperarista.

En la siguiente sección se profundizará en los elementos y semántica de la gramática GLG, describiendo en detalle cada uno de sus componentes y su funcionalidad.

4.4. Definición de un lenguaje de programación paralela mediante gramática de grafos

Desde su aparición en los años 60, las gramáticas de grafos (GG) se han aplicado en diferentes áreas donde es más fácil representar datos como diagramas o grafos que como cadenas de texto. Algunas de estas áreas son: especificación y desarrollo de software, reconocimiento de patrones, sistemas de bases de datos, especificación de tipos de datos, etc. En ciencias de la computación, la definición de un lenguaje visual paralelo mediante una gramática de grafos proporciona varias ventajas [BHD+95], entre las que se pueden destacar:

- Descripción visual de la concurrencia.
- Aprendizaje rápido del lenguaje.
- Descripción formal y visual del lenguaje.
- Identificación de programas (diagramas) que no pertenecen al lenguaje.
- Extensión de nuevas características al lenguaje.
- Generación de programas mediante un desarrollo dirigido por sintaxis.

La descripción visual de la concurrencia permite identificar las tareas que se realizan simultáneamente, lo cual es complicado en los lenguajes textuales. Además, tener un conjunto reducido de íconos como representación de las operaciones del lenguaje facilita un aprendizaje rápido, ya que una imagen es más fácil de recordar que una instrucción textual con sus parámetros.

El uso de una gramática formal, al igual que en los lenguajes textuales, permite comprender las reglas que definen el lenguaje. Por lo tanto, utilizar una gramática de grafos brinda una descripción formal y visual del lenguaje paralelo propuesto además de verificar si un programa cumple con las reglas de su lenguaje, lo que significa que se puede identificar aquellos programas (diagramas) que no pertenecen al lenguaje. Conocer las reglas de una gramática también facilita la posibilidad de ampliar las operaciones o funcionalidades del lenguaje en el futuro, al permitir la extensión de dichas reglas [Roq08].

Resumen

En este capítulo se ha presentado la teoría general de la gramática de grafos mediante el reemplazo de hiperaristas (GGRH) donde se establece un conjunto de reglas para reemplazar hiperaristas sobre hipergrafos. Proponer la gramática de GLG mediante una GGRH permitirá describir qué tipos de grafos pertenecen al lenguaje GLG y comprender sus alcances y limitaciones. Este formalismo es similar al de las gramáticas textuales, pero se utiliza para describir lenguajes visuales y diagramas.

La utilización de una GGRH para describir un lenguaje visual ofrece varias ventajas en términos de visualización de la concurrencia y desarrollo de programas válidos. Considerando estas ventajas, en el siguiente capítulo se define formalmente el lenguaje GLG propuesto y cada uno de sus elementos utilizando una gramática de grafos de reemplazo de hiperarista (GGRH) para describir el lenguaje GLG.

Capítulo 5

Lenguaje GLG

En este capítulo, se presenta nuestra propuesta GLG (*Graphic Language for GPUs*) junto con sus características y elementos, así como su gramática subyacente que define las reglas sintácticas del lenguaje. Es en este contexto que presentamos GLG como una solución innovadora a la complicada tarea de programar sobre tarjetas gráficas.

5.1. Características de GLG

El lenguaje GLG permite la construcción de programas visuales que son ejecutados en tarjetas gráficas. Las principales características se presentan a continuación:

- **Arrastrar y Soltar:** El entorno de desarrollo de GLG permite a los desarrolladores arrastrar y soltar elementos visuales para construir el flujo de trabajo de sus programas. Esto simplifica el proceso de diseño y agiliza el desarrollo, ya que los elementos visuales representan operaciones y funciones predefinidas en lugar de escribir código textual.

- **Programación Visual:** El lenguaje GLG se basa en la programación visual, lo que permite a los desarrolladores construir programas mediante la manipulación de íconos, asociados a un conjunto de opciones que se despliegan mediante una ventana de *pop-up*, por ejemplo, algunas propiedades

son: definir variables de comunicación, agregar código, definir bibliotecas, etc.

- **Reglas de producción:** Esta característica se basa en la definición de una gramática específica para el lenguaje GLG, que establece las reglas y restricciones sintácticas que deben cumplir los programas construidos en el entorno visual, de esta forma se asegura que los programas construidos en GLG sean correctos en cuanto a su estructura y sintaxis, evitando así la generación de programas inválidos o con errores sintácticos.

Las propiedades de los íconos, así como las reglas de producción de la gramática para GLG se presentan en las secciones siguientes.

5.2. Estructuras visuales e íconos de GLG

Los íconos de GLG se clasifican en: ícono de *Cabecera*, ícono de *Constantes*, ícono de *Variables de comunicación* e íconos de procesamiento (*CPU* y *GPU*). Cada ícono tiene un conjunto de propiedades que se pueden obtener y modificar haciendo clic sobre el mismo.

5.2.1. Ícono de *Cabecera*

El ícono de *Cabecera* permite seleccionar los archivos de cabecera (bibliotecas) que contienen subrutinas para realizar operaciones como: entrada y salida de datos, cálculos matemáticos, formato de hora y fecha, entre otros. Al hacer clic en el ícono, se muestra un menú emergente (*pop-up*) que presenta una lista de bibliotecas disponibles para que el usuario seleccione (Figura 5.1). El usuario puede explorar y seleccionar las bibliotecas relevantes para su programa a partir de esta lista. Las bibliotecas de cabecera proporcionan un conjunto de funciones predefinidas que pueden ser utilizadas en el programa para simplificar el desarrollo y aprovechar funcionalidades ya implementadas. Al elegir las bibliotecas adecuadas, el programador puede aprovechar la funcionalidad existente y ahorrar tiempo y esfuerzo en la implementación de ciertas operaciones comunes.

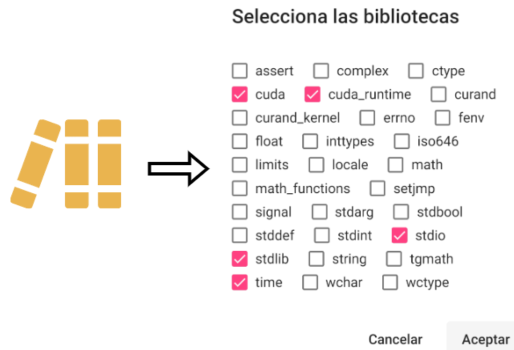


Figura 5.1: Ícono de *Cabecera* y *pop-up* correspondiente.

5.2.2. Ícono de *Constantes*

En el ícono de *Constantes* (Figura 5.2), se definen datos estáticos cuyos valores no pueden ser modificados durante la ejecución del programa. El usuario puede asignar un identificador y un valor a cada constante. Esto permite establecer valores fijos que serán utilizados en diferentes partes del programa y que no cambiarán a lo largo de su ejecución. Las constantes pueden ser números, cadenas de texto u otros tipos de datos que sean necesarios para el correcto funcionamiento del programa. Al definir las constantes en el ícono correspondiente, el programador tiene la posibilidad de organizar y gestionar de manera centralizada los valores constantes utilizados en su programa, lo que facilita su mantenimiento y modificación en caso de ser necesario en el futuro.

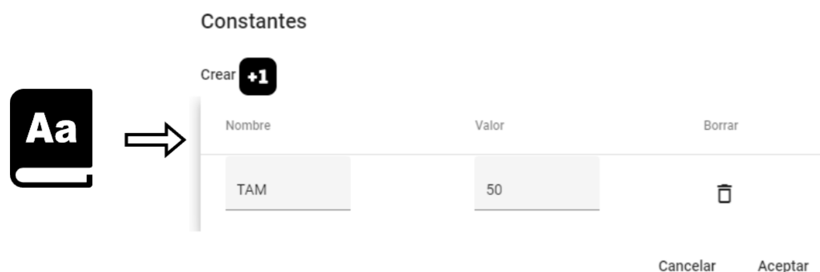


Figura 5.2: Ícono de *Constantes* y *pop-up* correspondiente.

5.2.3. Ícono de ***Variables de comunicación***

El ícono de *Variables de Comunicación* (Figura 5.3) en GLG proporciona una forma de crear variables que se utilizan para intercambiar datos entre el CPU y la GPU. Estas variables son fundamentales en el procesamiento paralelo, ya que permiten la transferencia de información entre estas unidades de procesamiento. El nombre en CPU es asignado por el usuario, mientras que la variable en GPU se crea por defecto. A diferencia de las constantes, en este componente el usuario tiene la capacidad de seleccionar el tipo de variable mediante una lista desplegable, lo que le permite elegir entre diferentes tipos de datos, como enteros, flotantes, caracteres, etc.

Además de seleccionar el tipo de variable, el usuario puede especificar la cantidad de elementos que contendrá la variable (arreglo unidimensional). Esto se realiza ingresando un número entero que representa la dimensión del vector de la variable. Dependiendo de la aplicación, se pueden crear variables con una sola dimensión o con múltiples dimensiones, lo que permite trabajar con estructuras de datos más complejas. Cabe destacar que el programador debe realizar la conversión de los datos con múltiples dimensiones a una dimensión y viceversa.

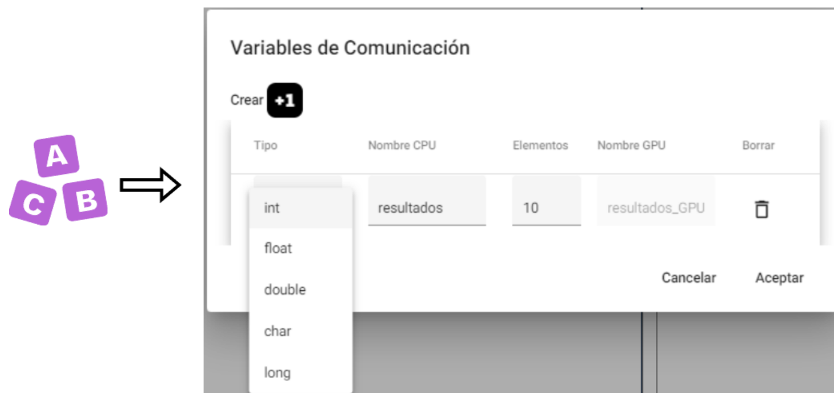


Figura 5.3: Ícono de *Variables de comunicación* y *pop-up* correspondiente.

5.2.4. Íconos de ***Procesamiento***

Los íconos de *Procesamiento* en GLG desempeñan un papel fundamental en la definición y ejecución de operaciones tanto en la CPU como en la GPU. Estos

íconos se dividen en dos categorías: íconos de procesamiento en la CPU y el ícono de procesamiento en la GPU.

Los íconos de procesamiento en la CPU permiten al usuario especificar el código que se ejecuta antes y después de un procesamiento en la GPU. El usuario puede indicar si una variable será utilizada para el envío o recepción de datos. Esto es importante en el contexto de la programación paralela, donde se requiere una comunicación eficiente entre la CPU y la GPU para sincronizar los datos y los cálculos.

En el ícono *CPU de entrada* (Figura 5.4) existe un apartado que se habilita al crear una o más variables de comunicación. Aquí el usuario selecciona aquellas que van a ser enviadas a la GPU. El código que es escrito en este *pop-up* es el que se ejecuta antes del llamado a la función que se ejecuta en la tarjeta gráfica.

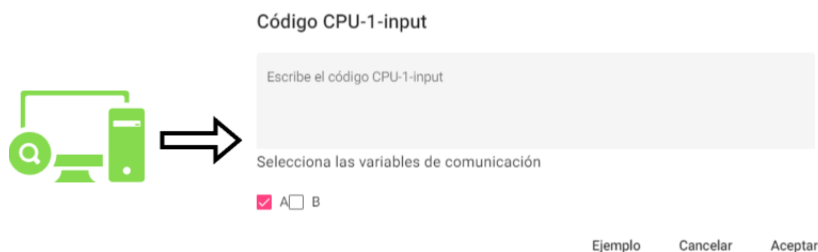


Figura 5.4: Ícono de procesamiento en la *CPU de entrada* y *pop-up* correspondiente.

Por otro lado, el ícono de procesamiento en la *GPU* (Figura 5.5) contiene el código que se ejecuta específicamente en la tarjeta gráfica. Aquí, el usuario tiene la capacidad de seleccionar el número de hilos a crear en la GPU. Además debe escribir el código de tal forma que se divida el trabajo de procesamiento entre los hilos.

Figura 5.5: Ícono de procesamiento en la *GPU* y *pop-up* correspondiente.

Finalmente, el ícono de la *CPU de salida* (Figura 5.6) desempeña un papel crucial en la sincronización de los hilos una vez que han finalizado la ejecución del kernel en la GPU. Además, este ícono se utiliza para recibir las variables seleccionadas por el usuario, las cuales provienen del dispositivo GPU. Esta sincronización y recuperación de datos son fundamentales para garantizar una correcta comunicación entre la CPU y la GPU, y lograr los resultados finales del procesamiento realizado.

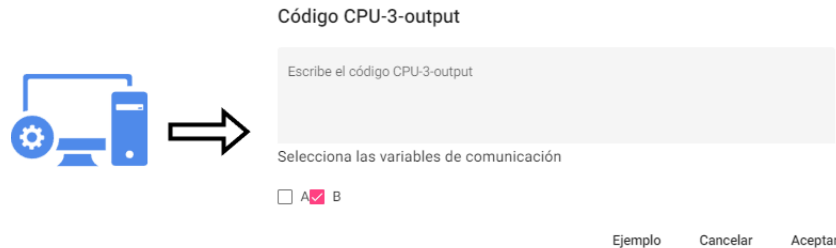


Figura 5.6: Ícono de procesamiento en la *CPU de salida* y *pop-up* correspondiente.

El ícono de *Salida* (Figura 5.7) desempeña un papel importante en la visualización y almacenamiento de los resultados del programa. Al hacer clic en este ícono, se abre un *pop-up* donde se muestra el resultado generado por el programa después de su compilación y ejecución. Este ícono permite observar en pantalla los resultados después de realizar la ejecución del programa o en su defecto, mostrar los errores de sintaxis o desbordamiento de memoria.

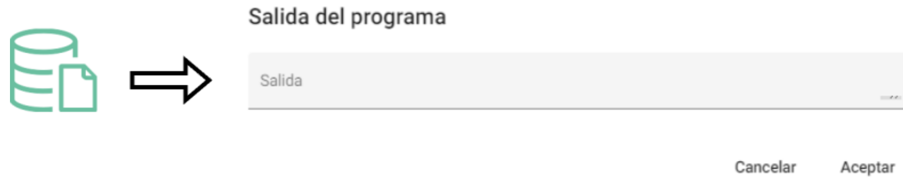


Figura 5.7: Ícono de *Salida* y *pop-up* correspondiente.

5.3. Gramática

En esta sección se presenta la gramática utilizada en GLG, la cual orienta al usuario como construir programas visualmente correctos, siguiendo las reglas definidas en la gramática. Esto proporciona una guía clara y precisa para la creación de programas en GLG. A continuación, se detallan las reglas de la gramática de GLG.

5.3.1. Elementos de la grámatica GLG

La gramática de GLG se define como una 4-tupla $\mathbf{G} = (N, \Sigma, R, S)$ donde:

- N representa el conjunto de hiperaristas (*no terminales*) del lenguaje; las cuales se muestran como palabras encerradas en un rectángulo.

  $N = \{$*Programa* y *Operaciones*$\}$. Las hiperaristas en GLG representan los elementos estructurales del programa gráfico.

- Σ representa los íconos coloreados que actúan como elementos terminales en la gramática. Estos íconos son las unidades básicas que componen el programa gráfico en GLG. Cada ícono tiene su propio significado y funcionalidad dentro del lenguaje.

- R muestra el conjunto de reglas de producción de GLG. Las reglas están identificadas por la etiqueta Ri, donde i es el número de regla. Cada regla establece las posibles sustituciones de un símbolo *no terminal* por una secuencia de símbolos terminales y/o no terminales.

- *S* denota el símbolo *no terminal* inicial que se utiliza para iniciar la generación del programa gráfico. El símbolo *Programa* define la estructura principal de un programa en GLG y actúa como punto de partida para la generación del programa completo.

La gramática de GLG proporciona las reglas y estructuras necesarias para construir programas gráficos válidos y coherentes en el lenguaje. A través de la combinación de hiperaristas e íconos, los usuarios pueden crear programas visualmente atractivos y funcionales en GLG, explorando las diversas posibilidades y combinaciones que ofrece la gramática.

5.3.2. Reglas de producción de la grámatica GLG

En la gramática de GLG mostrada en la Figura 5.8 el símbolo inicial es el símbolo *no terminal Programa* con la que se especifica la estructura principal de un programa en GLG, mientras que el símbolo *no terminal Operaciones* tiene 2 posibles sustituciones, que dan lugar a las reglas *R2* o *R3*.

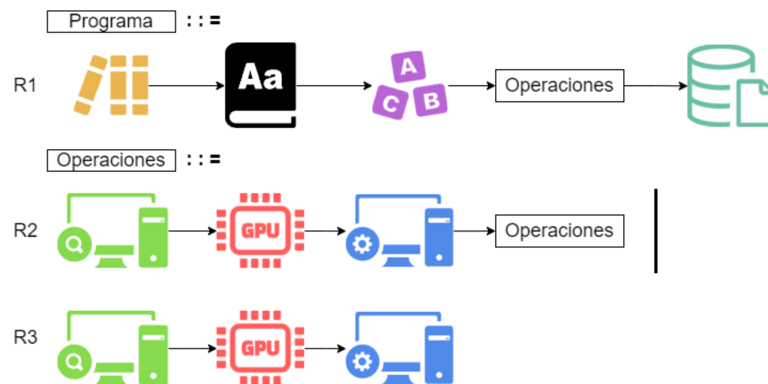
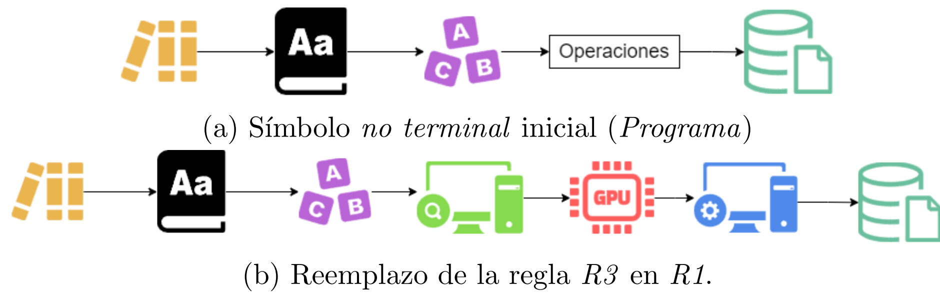


Figura 5.8: Reglas de producción de GLG.

Uno de los programas más simples que pueden ser construidos a partir de estas reglas de producción se muestra en la Figura 5.9a, donde la regla *R3* es sustituida por el símbolo *no terminal Operaciones* obteniendo un diagrama con puros símbolos terminales (Figura 5.9b).



(a) Símbolo *no terminal* inicial (*Programa*)

(b) Reemplazo de la regla *R3* en *R1*.

Figura 5.9: Programa de ejemplo construido con la gramática de GLG.

Resumen En este capítulo se describió el lenguaje GLG, sus íconos y su gramática. Los íconos en GLG se clasifican en íconos de *Cabecera*, *Constantes*, *Variables de comunicación* y procesamiento en la *CPU* y *GPU*. Cada uno de estos íconos tiene propiedades específicas y su inclusión en la gramática permite construir programas gráficos válidos en GLG.

Además, se presentaron las reglas de producción de la gramática de GLG, donde se especifican las reglas de producción para conectar los iconos afín de generar programas bien formados en GLG.

Capítulo 6

Ambiente GLG (GLG-E)

En este capítulo, se presenta el entorno de desarrollo GLG-E (*Graphic Language for GPUs-Environment*). Con el fin de comprender su funcionamiento completo, se describe su arquitectura, su interfaz gráfica, se detalla cómo se almacenan y procesan las propiedades de los íconos para construir el código CUDA y finalmente se muestra el funcionamiento del motor de ejecución remoto, que permite ejecutar programas en la nube.

Las principales características del entorno se presentan a continuación:

- **Entorno web:** El entorno de desarrollo de GLG se basa en una plataforma web, lo que permite a los desarrolladores acceder y utilizar el lenguaje desde cualquier dispositivo con conexión a internet y un navegador. Esto brinda flexibilidad y accesibilidad para el desarrollo de programas visuales en tarjetas gráficas.

- **Soporte para tarjetas gráficas NVIDIA:** El lenguaje GLG está diseñado específicamente para la programación sobre tarjetas gráficas NVIDIA. Proporciona un conjunto de herramientas y funciones para aprovechar al máximo el rendimiento de estas tarjetas en el procesamiento paralelo.

- **Procesamiento en la nube:** Se ofrece la posibilidad de realizar el procesamiento de los programas en la nube. Esto permite ejecutar los programas visualmente diseñados en tarjetas gráficas remotas, lo que proporciona una

mayor escalabilidad y flexibilidad en el procesamiento de grandes volúmenes de datos.

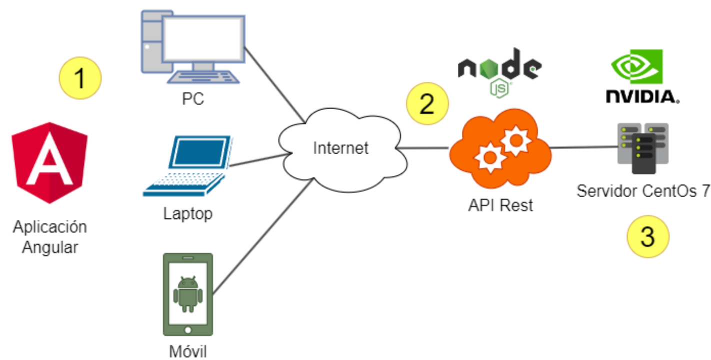
- **Comunicación mediante una API-Rest:** GLG utiliza una interfaz de programación de aplicaciones (API) basada en el protocolo REST para la comunicación entre el entorno de desarrollo y el servidor de ejecución en la nube. Esto facilita el intercambio de datos y resultados entre el entorno de desarrollo y el entorno de ejecución remoto.

6.1. Arquitectura de GLG

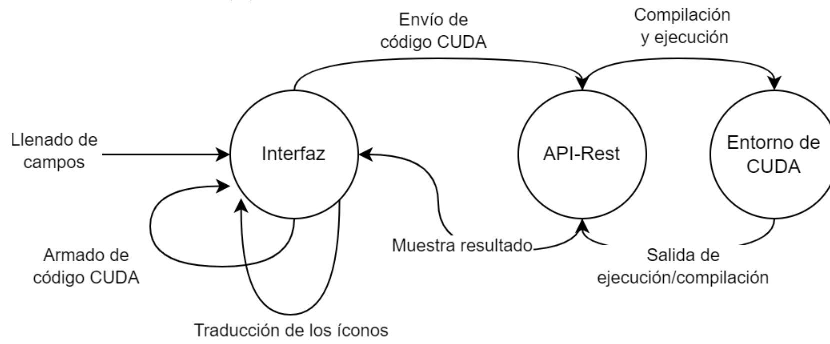
La arquitectura del entorno GLG-E se ilustra en la Figura 6.1a y consta de tres elementos principales: la interfaz (etiqueta 1), el motor de ejecución que se conforma por la API-Rest (etiqueta 2) y el entorno de CUDA remoto (etiqueta 3).

La Figura 6.1b muestra una máquina de estados en la que se muestra el flujo de ejecución e intercambio de datos entre los diferentes elementos de la arquitectura. En ella se muestra que la traducción y armado del código CUDA se realiza en la interfaz web, mientras que la API-Rest se encarga de enviar el código al entorno de CUDA remoto para realizar la compilación y ejecución. En caso de que haya errores de compilación o se haya realizado una ejecución exitosa, la salida de ambos procesos se regresa a la interfaz web para su visualización.

En las siguientes secciones se describen los elementos de la arquitectura de GLG.



(a) Arquitectura tecnológica de GLG-E.



(b) Máquina de estados del funcionamiento de GLG-E.

Figura 6.1: Arquitectura de GLG-E.

6.2. La interfaz

La interfaz proporciona la capacidad de generar y ejecutar programas en el lenguaje GLG. Los componentes de la interfaz se muestran en la Figura 6.2 mediante etiquetas circulares y se describen a continuación:

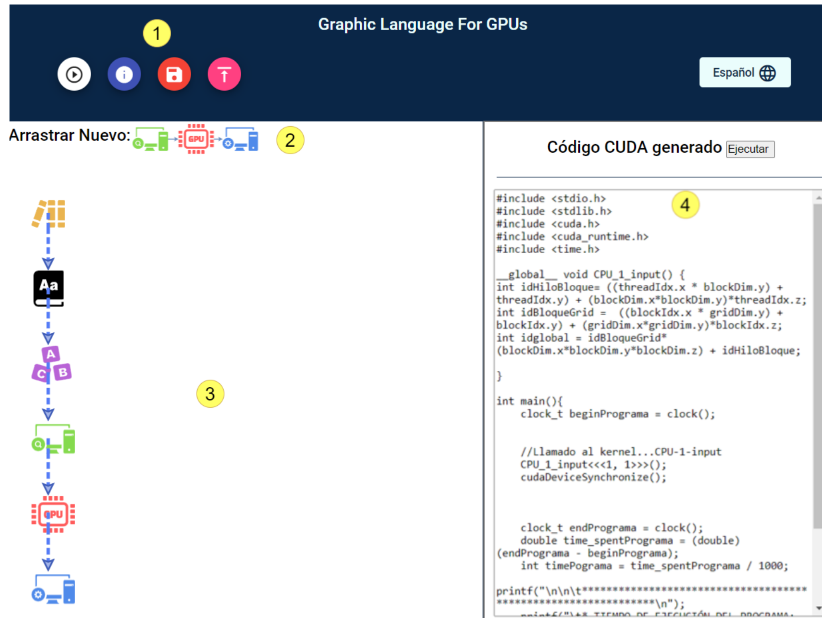


Figura 6.2: Interfaz de la aplicación GLG-E.

- El menú superior (*etiqueta 1*) permite realizar la traducción de los programas. Además, proporciona acceso a información detallada sobre la tarjeta gráfica, como el número de núcleos, el tamaño del *warp*, los hilos por bloque, entre otros. Al presionar el botón de *play*, se inicia la traducción y construcción del código, lo cual se explica en la siguiente sección.

- El componente *Nueva tarea de procesamiento* (*etiqueta 2*) permite agregar un nuevo bloque al flujo de trabajo. Para hacerlo, simplemente se arrastra el bloque deseado y se suelta sobre un ícono de *CPU de salida*. De esta manera, se garantiza que la gramática de GLG se respete, evitando la inserción del bloque si se suelta sobre otro ícono no válido.

- El área de trabajo (*etiqueta 3*) es el espacio donde se desarrollan los programas de GLG. En esta región, el usuario tiene la capacidad de seleccionar y completar las propiedades de los diferentes íconos, como los íconos de *Cabecera*, *Constantes* y *Variables de comunicación*. En el caso de los íconos de *procesamiento*, las instrucciones se escriben en lenguaje C para definir las operaciones y algoritmos que se llevarán a cabo en el programa.

- Finalmente, el código generado después de la traducción se muestra en el área de texto (*etiqueta 4*). Para ejecutar el código, se presiona el botón *ejecutar* y GLG-E envía el código al servidor para su compilación y ejecución.

6.2.1. Traducción

La traducción consiste en armar las sentencias de código C y CUDA a partir de las propiedades que ha seleccionado y escrito el usuario. Al dar aceptar al diálogo de un ícono, se comienza a generar el código y se guarda en una lista de objetos. A continuación se explica la forma en que se realiza la traducción de los datos.

Código de cabecera: A partir del ícono de *Cabecera* se obtiene una lista con los nombres de las bibliotecas seleccionadas por el usuario. Cada nombre de la biblioteca se inserta en la sintaxis adecuada para construir línea por línea. Esto se ilustra en la Figura 6.3, donde se muestra un ejemplo de cómo se genera el código de las bibliotecas.

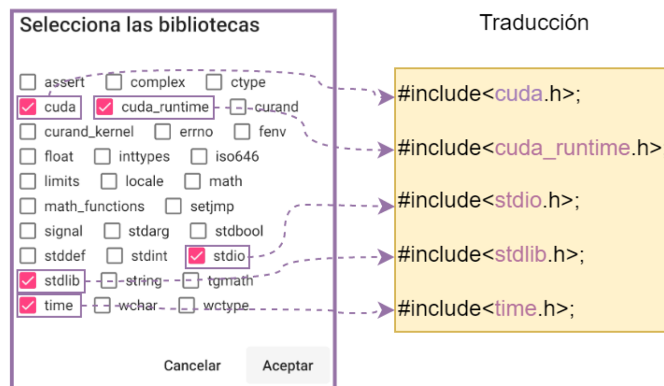


Figura 6.3: Ejemplo de traducción de las propiedades del ícono de *Cabecera*.

Este enfoque permite que el usuario seleccione las bibliotecas que necesita para su programa y que el código CUDA resultante incluya automáticamente las instrucciones correspondientes. De esta manera, se simplifica el proceso de escritura del código CUDA y se asegura que todas las bibliotecas necesarias estén presentes en el programa generado.

Código de constantes: Para cada constante creada, tanto el campo *nombre* como *valor*, se utilizan para generar la sintaxis adecuada en el código CUDA. Esto se ilustra en la Figura 6.4, donde se muestra un ejemplo de cómo se realiza esta traducción.

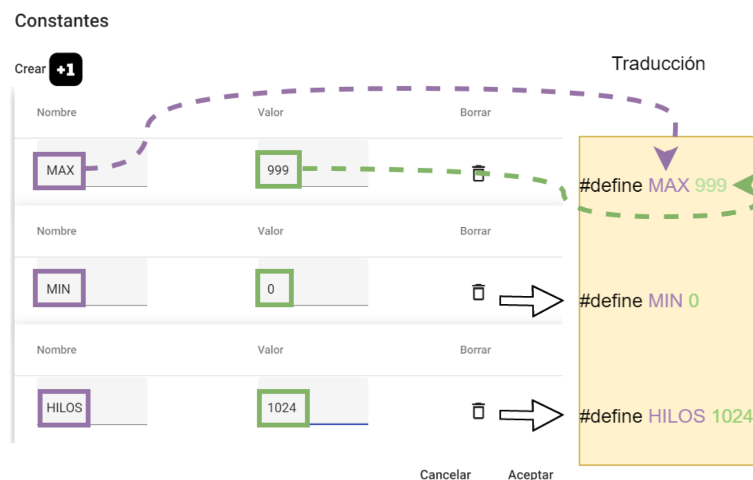


Figura 6.4: Ejemplo de traducción de las propiedades del ícono de *Constantes*.

El proceso de traducción garantiza que las constantes definidas por el usuario sean incluidas correctamente en el código CUDA generado. De esta manera, el usuario puede establecer los valores de las constantes de forma intuitiva en GLG-E, y el código resultante reflejará esos valores en la definición de las constantes en CUDA.

Código para la definición de las variables de comunicación: Para cada variable, se tienen dos identificadores (nombres): uno para la CPU y otro para la GPU. También, se almacena el tipo y la dimensión de la variable. Con esta información, se generan sentencias de código que se ubican en diferentes partes del programa. En primer lugar se genera el código que define las variables y reserva espacio de memoria en el entorno de la CPU y GPU, permitiendo su uso en el *host* y el *device* (Figura 6.5).

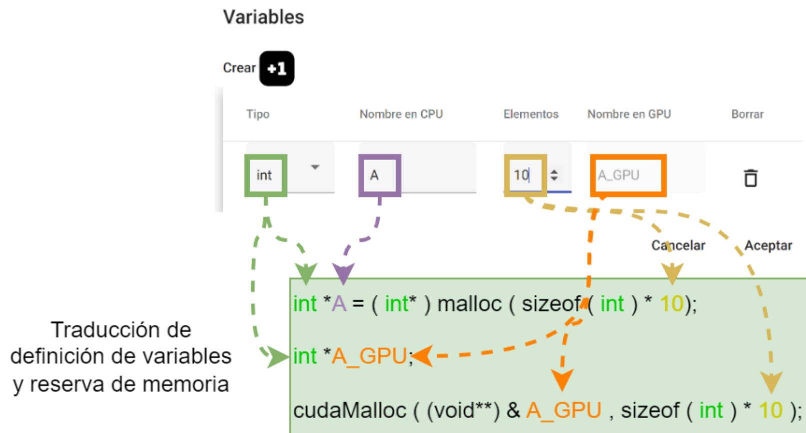


Figura 6.5: Ejemplo de traducción para la definición de las variables de comunicación y su reserva de memoria en la CPU y GPU.

Código para el envío y recepción de las variables de comunicación: Las instrucciones que permiten realizar la transferencia de datos entre el *host* y el *device* se construyen a partir de los datos del ícono de *Variables de comunicación* como se muestra en el ejemplo de la Figura 6.6. Estas instrucciones garantizan que los datos sean accesibles en ambos dispositivos y que los resultados del procesamiento en la GPU puedan ser utilizados en la CPU de manera adecuada.

Cabe mencionar que existe el parámetro *cudaMemcpyDefault* que determina el origen y destino del copiado de los datos de forma automática. Sin embargo, GLG crea explícitamente la instrucción con las direcciones de copiado *cudaMemcpyHostToDevice* y *cudaMemcpyDeviceToHost*. Esto se debe a que en algunos sistemas donde no se admite el direccionamiento virtual unificado, la decisión sobre si los datos deben ser enviados a la CPU o a la GPU no puede ser resuelta de manera automática.

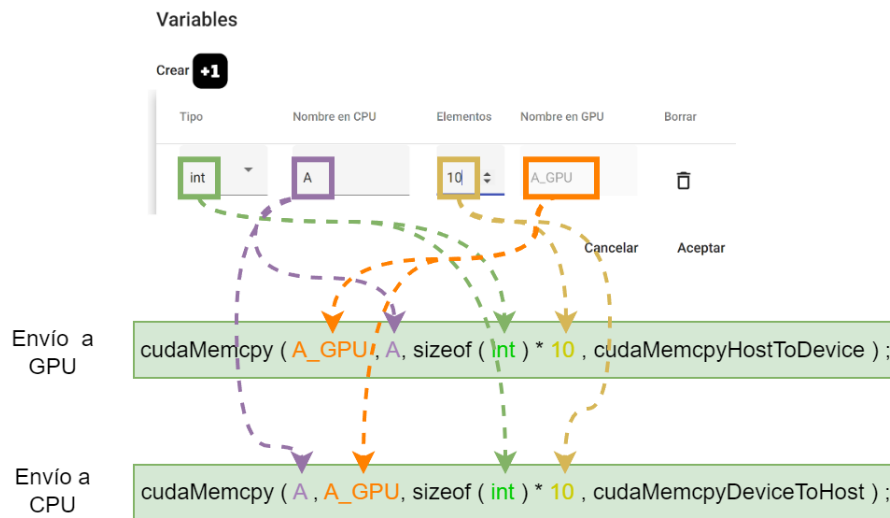


Figura 6.6: Ejemplo de traducción de las propiedades del ícono de *Variables de comunicación*.

Código de la definición y llamado al kernel: La definición del kernel, implica especificar el nombre de la función y los argumentos necesarios para su ejecución. Estos últimos corresponden a las variables de comunicación que son enviadas desde la CPU, por lo que los argumentos en la definición del kernel se construyen a partir del identificador y el tipo de la variable. Esto permite que los datos sean transmitidos correctamente desde la CPU a la GPU al momento de ejecutar el kernel. En la Figura 6.7 se muestra un ejemplo de la definición de un kernel usando 2 variables de comunicación (Figura 6.7a) y su respectiva invocación (Figura 6.7b).

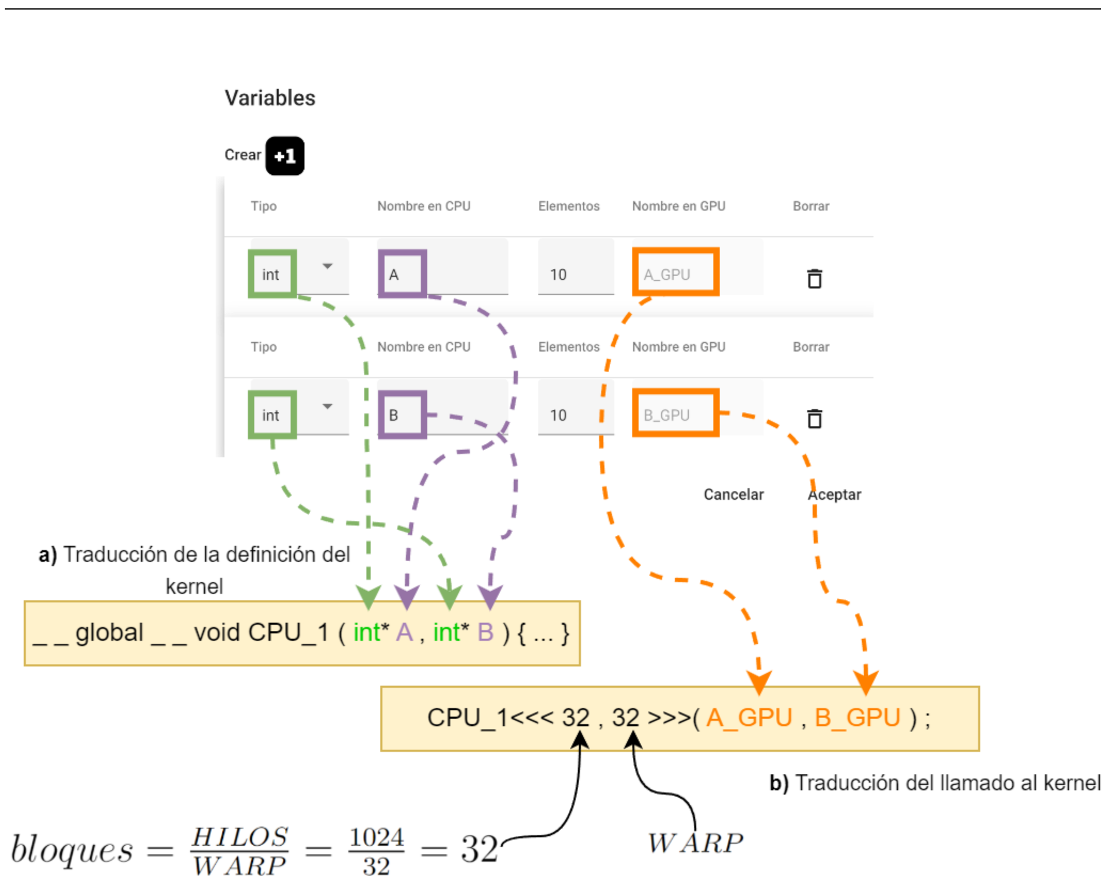


Figura 6.7: Ejemplo de traducción de la definición, llamado al kernel y cálculo del número de bloques.

En la traducción del llamado al kernel se establece el número bloques e hilos que serán creados en el *device*. Debido a que el único parámetro que ingresa el usuario en el ícono de *GPU* es el número de hilos, se realiza un cálculo para obtener el número de bloques con base en el tamaño del *warp*. Por ejemplo, la tarjeta gráfica utilizada para ejecutar los programas de GLG-E tiene un tamaño del *warp* de 32. Si el usuario ingresa el valor de 1024 para el número de hilos, el cálculo sería el siguiente: $bloque = \frac{HILOS}{WARP} = \frac{1024}{32} = 32$.

Los argumentos del llamado al kernel corresponden a las variables que son enviadas a la GPU (Figura 6.7b). Estos valores son tomados del ícono de *Variables de comunicación*, los cuales son creados de forma automática con base en el nombre en la CPU.

A continuación se detalla la estructura de datos que se utiliza para almacenar cada una de las propiedades y sentencias traducidas a lenguaje C y CUDA.

6.2.2. Almacenamiento interno: Listas de de objetos

GLG-E cuenta con dos listas de objetos para almacenar los datos que permiten el correcto funcionamiento de la aplicación web. Una lista (*Propiedades de íconos*) almacena los datos y su traducción. Otra lista (*Líneas*) es utilizada para guardar la relación entre los íconos para dibujar las líneas. A continuación se describe cada una de las listas.

1. Lista de objetos: *Propiedades de íconos*

Cada ícono está asociado a una clave con el mismo nombre del componente y todos ellos tienen claves adicionales llamadas *idPadre* e *idHijo*. Estas claves representan los identificadores de los componentes padre e hijo, respectivamente, a los cuales el componente en cuestión está conectado. De esta manera, se puede seguir el flujo de trabajo y establecer las relaciones entre los distintos componentes. A continuación se explica el contenido de cada clave asociada a las propiedades de los íconos (Figura 6.8).

- **cabecera**: Esta clave contiene una lista de bibliotecas. Cada biblioteca tiene un valor de verdadero o falso, dependiendo de si el usuario la ha seleccionado. Las sentencias generadas se almacena en la clave *código*.

- **constantes**: Las constantes creadas se guardan en una lista junto con sus identificadores y valores correspondientes, los cuales se almacenan en la clave *lista*. La clave *código* guarda las sentencias utilizadas para definir constantes en lenguaje C.

- **variables**: La clave *lista* contiene el conjunto de variables de comunicación creadas por el usuario. Aquí se guardan los atributos de la variable, como el nombre en la CPU y GPU, el tipo y la dimensión (tamaño). Este componente almacena tres tipos de código:

  - *Definición*: El código que define la variable y reserva memoria tanto en la CPU como en la GPU.
  - *Envío*: El código utilizado para copiar las variables de la CPU a la GPU.

- *Recepción*: El código utilizado para enviar las variables de comunicación de vuelta a la CPU.

- **CPU de entrada y salida**: Estas claves contienen la lista de variables de comunicación seleccionadas por el usuario para ser enviadas o recibidas entre la CPU y la GPU. Con esta lista, es posible crear el campo llamado *argsLlamado*, que guarda los parámetros que se pasan como argumento cuando se realiza una llamada al kernel. El campo *argsMetodo* se utiliza para construir la función del kernel, ya que además del nombre de la variables, se deben especificar los tipos. El código previo y posterior a la ejecución del kernel se almacena en el campo *código* de la CPU de entrada y salida respectivamente.

- **GPU**: el campo *código* contiene las sentencias que se ejecutarán en la tarjeta gráfica y que son escritas por el usuario en lenguaje C. También almacena el número de hilos con los que se ejecutará el kernel.

- **output**: Almacena el resultado de ejecución del programa.

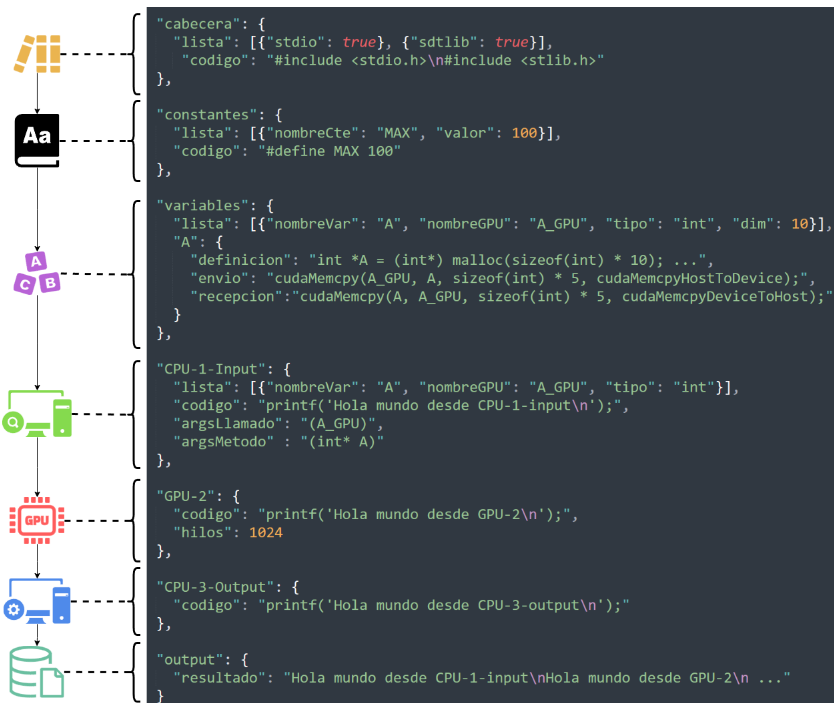


Figura 6.8: Ejemplo de la estructura del almacenamiento de datos de los íconos de GLG-E.

2. Lista de objetos: Líneas

La segunda lista de objetos almacena los datos que conectan visualmente a los íconos con líneas animadas. Cada línea tiene un conjunto de propiedades que definen el color de la línea y su grosor, además de determinar si es unidireccional o bidireccional. También se establece una animación de movimiento para las líneas, la cual también se guarda en esta estructura (Figura 6.9). Estas propiedades son definidas de forma estática al iniciar la aplicación.

Es importante conocer la relación entre el componente padre e hijo para que las líneas sigan el movimiento del ícono arrastrado. Por esta razón, también se almacena el nombre del componente padre e hijo, para que al momento de insertar un nuevo bloque de procesamiento, se realicen las conexiones correspondientes con los nuevos íconos.

Figura 6.9: Ejemplo de la estructura de almacenamiento de las propiedades de las líneas.

En caso de arrastrar un ícono, se realiza una actualización de las líneas asociadas para que sigan la nueva posición del ícono desplazado. Esta información se encuentra en la lista de objetos, permitiendo un acceso rápido y eficiente durante la manipulación de los íconos en el entorno de GLG.

Como se ha descrito en esta sección, cada uno de los íconos crea su código correspondiente en C y CUDA al momento de realizar la traducción. Una vez que el usuario presiona el botón *construir* se comienza a generar el programa como se explica en el siguiente apartado.

Construcción del código completo

A medida que se accede a los campos para obtener las sentencias traducidas de los íconos, se va concatenando el código resultante en una cadena única para crear la estructura que se muestra en el ejemplo de la Figura 6.10. Además, se incluyen fragmentos de código adicionales, como mecanismos de sincronización, medición del tiempo de ejecución, liberación de memoria en CPU y GPU, entre otros, para mejorar la funcionalidad del programa (sentencias con el recuadro amarillo).

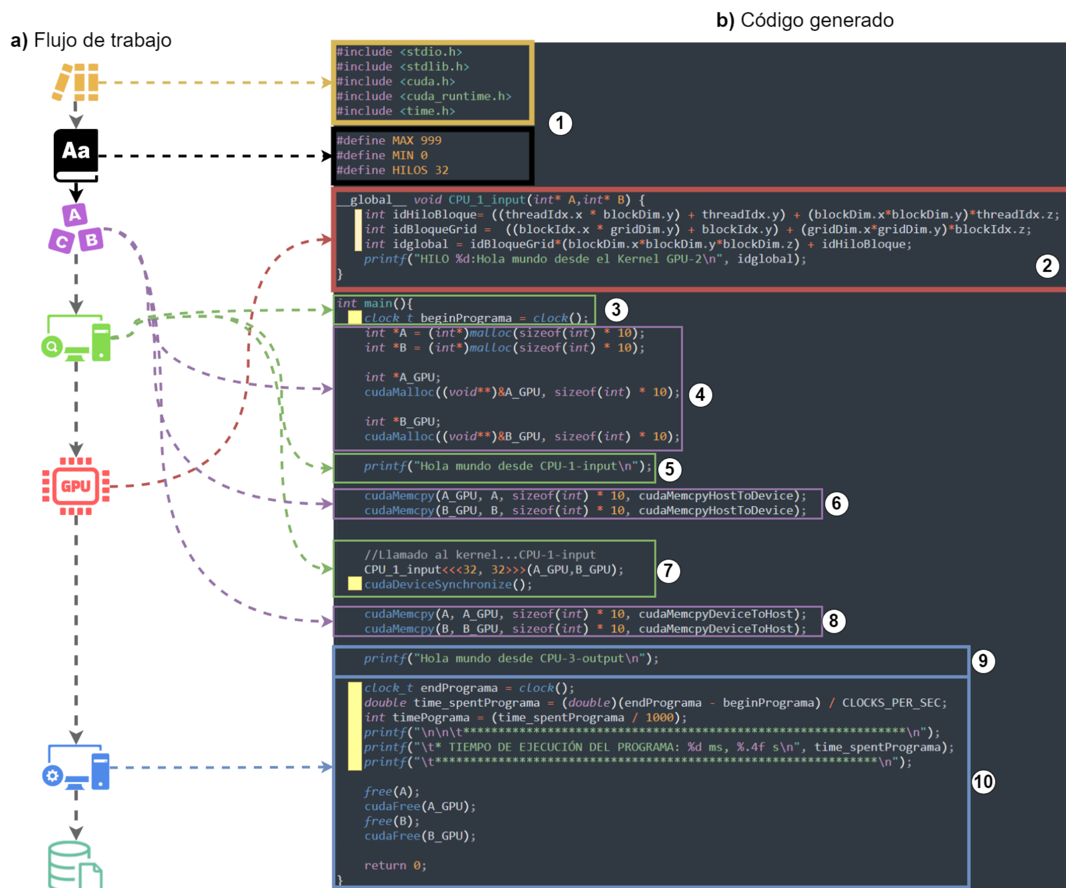


Figura 6.10: Ejemplo del código construido a partir de las propiedades de los íconos de GLG-E.

El orden para concatenar el código está señalado con círculos blancos y cada uno de estos puntos se detalla a continuación:

1. Se recopila la información relacionada con las bibliotecas y las constantes para crear el código correspondiente en la sección de cabecera y constantes del programa.

2. Se procesan las propiedades del ícono relacionado con la GPU, donde se completa la sintaxis de la función del kernel con base en los argumentos construidos en las variables de comunicación y se inserta el código escrito por el usuario.

3. Se construye el método *main*.

4. Se procesan las variables de comunicación entre la CPU y la GPU, insertando las instrucciones necesarias para su definición y reserva de memoria.
5. Se inserta el código escrito por el usuario de la *CPU de entrada*.
6. Se insertan las instrucciones de envío hacía la GPU.
7. Se genera el llamado al kernel y se establece el mecanismo de sincronización.
8. Se realiza la recepción de los datos provenientes de la GPU.
9. Se inserta el código escrito por el usuario en el componente *CPU de salida*.
10. Se inserta el código para medir el tiempo de ejecución y liberación de memoria.

Cabe mencionar que en el kernel se genera el código necesario para que los hilos puedan calcular un valor entero único con base en los identificadores del hilo y las dimensiones del bloque y malla. Este valor funciona como un **identificador global** de hilo, de esta manera el usuario puede utilizar *idglobal* como palabra reservada de GLG-E para distribuir la carga de trabajo entre los hilos.

El proceso de armado del código final garantiza que todas las instrucciones y elementos necesarios para la ejecución del programa estén presentes y en el orden adecuado. Una vez generado el código completo, se encuentra listo para ser procesado por el motor de ejecución.

6.3. Motor de ejecución

El motor de ejecución es responsable de enviar el código CUDA al servidor, ejecutar el programa y regresar el resultado al entorno de desarrollo para su visualización posterior. Está compuesto por dos partes: la API-Rest y el entorno de CUDA remoto.

6.3.1. La API-Rest

La API-Rest define un conjunto de servicios que permiten el intercambio de información entre el entorno de desarrollo y el entorno de CUDA remoto (cliente-servidor). La información se envía utilizando el protocolo HTTP y en formato JSON, además es posible enviar metadatos como el tipo de codificación, credenciales de autenticación o restringir las solicitudes a un dominio específico.

El servicio principal se encarga de enviar el código CUDA generado desde el entorno web al servidor en formato JSON. Una vez que la información es recibida en el servidor, se crea un archivo con extensión *.CU* (formato de CUDA) para almacenar el código y realizar la compilación con el comando NVCC (*NVIDIA CUDA Compiler*). El archivo binario es ejecutado y la salida del programa se regresa al entorno visual mediante el formato JSON como respuesta de la API-Rest.

Otros servicios que ofrece la API-Rest son los siguientes:

- **Verificación de conexión:** Permite verificar la conectividad con el servidor de la API-Rest para asegurar un acceso adecuado a los servicios.
- **Obtención de características de la tarjeta gráfica:** Permite obtener información detallada sobre la tarjeta gráfica instalada, como el modelo, la memoria, la velocidad de reloj, entre otros.
- **Monitoreo de procesos:** Proporciona la capacidad de monitorear y supervisar los procesos en ejecución en la tarjeta gráfica. Esto puede incluir información sobre el uso de la GPU, la utilización de la memoria, entre otros parámetros relevantes.
- **Descarga de códigos de ejemplo:** La API-Rest ofrece la posibilidad de descargar códigos de ejemplo para facilitar el aprendizaje y la implementación de aplicaciones con GLG-E.

Estos servicios permiten a los desarrolladores interactuar con la tarjeta gráfica de manera eficiente y obtener información relevante sobre su funcionamiento. La Figura 6.11 ilustra los diferentes servicios que ofrece la API-Rest.

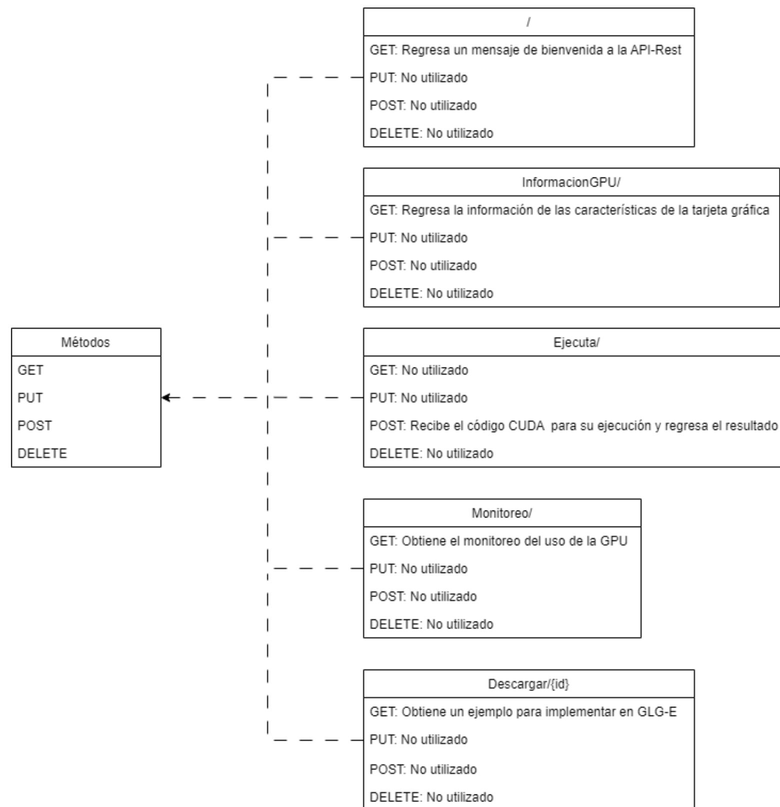


Figura 6.11: Servicios de la API-Rest de GLG-E.

6.3.2. El entorno de CUDA remoto

El entorno de CUDA remoto desempeña un papel fundamental en la ejecución de programas desarrollados con GLG. Esta característica permite una ejecución eficiente y controlada de los programas en CUDA de manera remota, junto con la recuperación de los resultados generados.

En este contexto, se establece una conexión entre el entorno de desarrollo web y un servidor con sistema operativo Linux CentOS 7.9. El servidor cuenta con una tarjeta gráfica NVIDIA *GeForce GTX 1080 Ti* con *CUDA Driver Version 11.6*, la cual es la última versión de CUDA actualmente (febrero, 2022).

Esta tarjeta gráfica es el motor de procesamiento para los programas creados utilizando GLG. La ejecución remota proporciona beneficios clave, como la utilización eficiente de recursos y la capacidad de ejecutar aplicaciones complejas sin

la necesidad de que el usuario cuente con una tarjeta gráfica en su sistema local.

Para aprovechar esta funcionalidad, es esencial asegurarse de que el servidor esté configurado adecuadamente. Esto incluye tener los controladores y bibliotecas de CUDA instalados en el servidor para realizar la compilación y ejecución de programas.

Resumen

En resumen, el capítulo sobre el entorno GLG proporciona una visión general de la herramienta y explica el proceso de traducción y construcción del código CUDA mediante la concatenación de las sentencias traducidas a partir de las propiedades de cada ícono. Además se mencionan los elementos adicionales que se insertan en el código, como mecanismos de sincronización y medición del tiempo de ejecución y liberación de memoria.

Capítulo 7

Evaluación y resultados

En el capítulo anterior se presentó GLG-E, ambiente que nos permite desarrollar programas paralelos utilizando la tecnología CUDA. Este capítulo se enfoca en la evaluación GLG y su ambiente mediante una comparación cualitativa y de desempeño.

7.1. Comparación cualitativa

La comparación cualitativa se divide en dos, una comparación cualitativa del lenguaje GLG respecto al desarrollo en CUDA de forma textual, ya que es una de las formas más populares de programar sobre tarjetas gráficas. La segunda es una comparación cualitativa respecto a los lenguaje visuales encontrados en la literatura.

Ambas comparaciones se basan en los criterios de expresividad, aplicabilidad, nivel de interacción y portabilidad explicadas en la Sección 2.3. A su vez, cada característica cuenta con criterios que sirven para evaluar de forma cuantitativa la característica correspondiente y así asignar una puntuación. A continuación se listan estos criterios:

- **Expresividad**:

    1. Cantidad de íconos

2. Íconos de tareas secuenciales
3. Íconos de estructuras de selección
4. Íconos de estructuras de repetición
5. Íconos de operaciones de entrada y salida (I/O)

- **Aplicabilidad**:
  1. Gramática
  2. De propósito general
  3. Patrón SPMD (*Single Program Multiple Data*)
  4. Patrón MPMD (*Multiple Program Multiple Data*)

- **Nivel de interacción**:
  1. Generación de código base (bibliotecas, *main*, etc.)
  2. Elementos *drag-and-drop*
  3. Encapsulamiento de íconos
  4. Re-ordenamiento de íconos
  5. Zoom al área de trabajo

- **Portabilidad**:
  1. Transparencia en la **ejecución** con la tarjeta gráfica
  2. Transparencia en la **localización** de la tarjeta gráfica

7.1.1. Evaluación respecto a programación textual

Para llevar a cabo una comparación entre CUDA y GLG, se consideran todos los aspectos de cada característica que ambos lenguajes abordan.

En relación a GLG, en términos de **Expresividad**, el lenguaje dispone de un número significativo de íconos para la creación de programas (7 íconos), incluyendo íconos para tareas secuenciales y operaciones de E/S (3 puntos para expresividad). En el aspecto del **Nivel de Interacción**, GLG proporciona generación

de código base, funcionalidades de arrastrar y soltar (elementos *drag-and-drop*) y re-organización de íconos (3 puntos para nivel de interacción). En lo que respecta a la **Portabilidad**, GLG ofrece transparencia en la ejecución y detección de la tarjeta gráfica (2 puntos para portabilidad). En cuanto a la **Aplicabilidad**, tanto CUDA como GLG poseen gramáticas definidas, permiten la creación de programas de propósito general y siguen el patrón SPMD (3 puntos para aplicabilidad).

Esta puntuación se simplifica en la Tabla 7.1, donde la primera columna hace referencia a las características de los lenguajes visuales. Las columnas siguientes detallan las propiedades cumplidas tanto por CUDA como por GLG.

Característica	CUDA	GLG
Expresividad		***
Aplicabilidad	***	***
Nivel de Interacción		***
Portabilidad		**

Tabla 7.1: Tabla comparativa de características de CUDA contra GLG.

En la tabla anterior se observa que CUDA ofrece un bajo nivel de expresividad, ya que no cuenta con íconos. Esto afecta directamente al nivel de interacción. GLG, por otro lado, simplifica la programación mediante un enfoque visual, lo que lo hace más accesible para usuarios sin experiencia en CUDA. Además, GLG genera código automáticamente y facilita la escritura de instrucciones complejas, como la creación y el envío de variables entre la CPU y la GPU. También permite al usuario modificar el código generado antes de ejecutarlo, en caso de que el usuario tenga un conocimiento más detallado de CUDA para optimizar o mejorar las instrucciones.

Además de estas diferencias, GLG ofrece otras ventajas sobre la programación textual en CUDA:

- **Expresividad visual:** Permite representar visualmente conceptos y estructuras, lo que facilita la construcción de los programas.

- **Facilidad de aprendizaje:** GLG al ser un lenguaje visual facilita el proceso de aprendizaje, ya que no se requiere un conocimiento profundo de la sintaxis y la gramática de CUDA.

- **Mayor productividad:** GLG acelera el proceso de desarrollo mediante acciones de arrastrar y soltar sobre elementos visuales. Lo anterior para construir la lógica y funcionalidad del programa, lo que reduce la necesidad de escribir código manualmente.

- **Portabilidad**: GLG es una capa de abstracción sobre CUDA, por lo que el entorno de desarrollo no depende de la plataforma o navegador en que se ejecute para generar código CUDA.

7.1.2. Evaluación respecto a lenguajes visuales

De forma análoga en la que obtiene la puntuación para CUDA y GLG, en la Tabla 7.2 muestra una comparación entre GLG y los lenguajes visuales revisados en el Capítulo 3. La primera columna representa las características de los lenguajes visuales, como la expresividad, la aplicabilidad, el nivel de interacción y la portabilidad. Las columnas restantes representan los diferentes lenguajes visuales evaluados, donde se asigna la puntuación con base en las propiedades que cumple cada característica, en donde una puntuación más baja indica un rendimiento deficiente en esa característica.

Característica	CUDABlock	OpenCLGen	GPUBlocks	GLG
Expresividad	*****	*	*****	***
Aplicabilidad	***	*	***	***
Nivel de Interacción	*		*	***
Portabilidad	*	**	*	**

Tabla 7.2: Tabla comparativa de los lenguajes visuales contra GLG.

Como se muestra en la tabla anterior, tanto CUDABlock como GPUBLock

tienen una buena expresividad al proporcionar un entorno visual para el desarrollo de programas, donde ofrecen más de 20 íconos enfocados en operaciones específicas de CUDA. Sin embargo, esta abundancia de íconos podría incrementar la curva de aprendizaje del lenguaje visual y resultar contraproducente para usuarios con bajo conocimiento del lenguaje. En contraste, GLG cuenta con un total de 7 íconos que inicialmente pueden generar la estructura básica del código para un programa de CUDA.

En cuanto a aplicabilidad, OpenCLGen está enfocado a resolver problemas de propósito específico, lo cual limita la cantidad de problemas que puede resolver, a diferencia de CUDABlock, GPUBlock y GLG , que tienen la capacidad de abordar una amplia variedad de problemas en diferentes campos de la programación, como la simulación de sistemas complejos, el procesamiento de datos en paralelo o la computación científica. Una de las aplicaciones en este último campo se puede observar en la astrofísica, donde se utilizan algoritmos paralelos en tarjetas gráficas para implementar métodos numéricos [Dav19].

Por otro lado, el entorno de GLG ofrece un alto nivel de interacción al proporcionar una interfaz donde los usuarios pueden arrastrar, soltar, re-ordenar íconos y establecer conexiones entre ellos, además de generar código base; a diferencia de CUDABlock, GPUBlock y OpenCLGen que carecen de elementos re-ordenables y generación de código base.

Por último, GLG-E retoma los puntos más fuertes de OpenCLGen en cuanto a portabilidad, ya que ambos pueden ejecutar los programas de manera transparente. Esto es posible al utilizar una API-Rest para el intercambio de datos entre el entorno remoto de CUDA y la aplicación web.

7.2. Comparación de desempeño

La comparación de desempeño se realiza mediante la solución de dos problemas (casos de prueba) usando la programación textual en CUDA y mediante el desarrollo visual con GLG. En esta sección se describen cada uno de las problemáticas, la solución paralela mediante GLG y por último, se muestra una comparación de tiempos de ejecución.

7.2.1. Casos de prueba

Los casos de prueba corresponden a la solución paralela de la generación de un diagrama Voronoi con 10 regiones (17920 x 17920 puntos asignados en alguna región) y el cálculo del área bajo la curva con 5 millones de particiones. A continuación se describen cada uno de problemas.

7.2.1.1. Diagramas de Voronoi

Estos diagramas se utilizan para dividir un espacio en regiones adyacentes, donde cada región está asociada a un punto específico llamado generador [GOY89]. En la Figura 7.1 se muestra la representación gráfica de un diagrama de Voronoi.

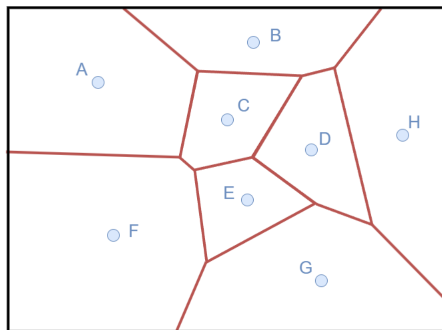


Figura 7.1: Diagrama de Voronoi en dos dimensiones.

Los diagramas de Voronoi tienen diversas aplicaciones en áreas como la robótica, donde se utilizan para planificar el recorrido de un robot evitando obstáculos. También se emplean en el modelado 3D, donde se utilizan para la generación de objetos creados con polígonos. Además, estos diagramas son útiles en la optimización de modelos 3D, ya que permiten analizar la distribución espacial de puntos o datos y optimizar procesos en función de esta distribución [Rod15].

7.2.1.2. Área bajo la curva

En esta problemática se utiliza la técnica de las sumas de Riemann para aproximar el área bajo la curva de una función $f(x)$ en el intervalo $[a, b]$. Las sumas de Riemann son una herramienta matemática que permite estimar áreas

mediante rectángulos. Para calcular la base de un rectángulo se divide el intervalo para obtener particiones muy pequeñas (Δx). Para el cálculo de la altura, se evalúa $a + i * \Delta x$ en la función, para cada valor de $i = 0, 1, 2, ..., n - 1$, donde n corresponde al número de particiones.

Finalmente, se calcula el área de cada rectángulo y se suman todas las áreas para obtener una aproximación del área total bajo la curva de la siguiente manera:

$$\acute{a}rea \approx \Delta x[f(a) + f(a + \Delta x) + f(a + 2\Delta x) + ... + f(b - \Delta x)] \qquad (7.1)$$

En la Figura 7.2 se muestra una representación gráfica de un ejemplo, realizando la partición del intervalo y los rectángulos correspondientes a cada subintervalo.

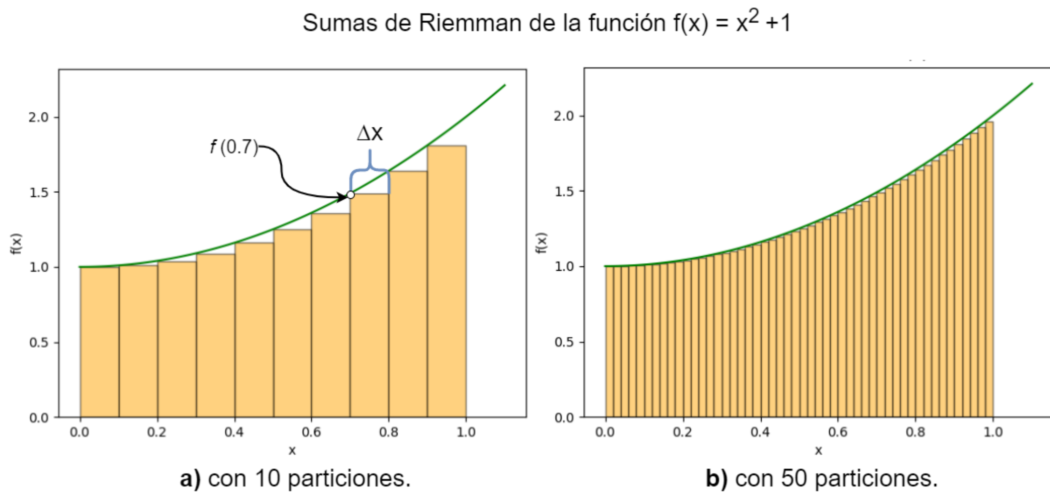


Figura 7.2: Ejemplo de sumas de Riemman para la aproximación del área bajo la curva.

7.2.2. Implementación de los problemas en GLG

A continuación se describe la solución paralela para cada problema mencionado en la sección anterior, así como su implementación en GLG.

7.2.2.1. Diagramas de Voronoi

La solución paralela para la generación de los diagramas Voronoi se basa en una matriz bidimensional de tamaño $n \times m$, donde se asocia un conjunto de filas a cada hilo para distribuir la carga de trabajo.

La región se obtiene de calcular la distancia euclidiana del elemento en cuestión contra todos los generadores, y se registra la menor distancia obtenida. La región a la que pertenece el elemento se determina según el generador asociado a esa menor distancia. Posteriormente, el hilo asigna el valor del generador a ese elemento de la matriz, lo cual indica a qué región pertenece.

La Figura 7.3 muestra un caso particular de una matriz de 10 x 10 y 5 regiones. En este ejemplo, solo se consideran 2 hilos (Figura 7.3a) para ejemplificar la distribución de filas. La Figura 7.3b muestra el calculo de distancia euclidiana para un elemento de la matriz.

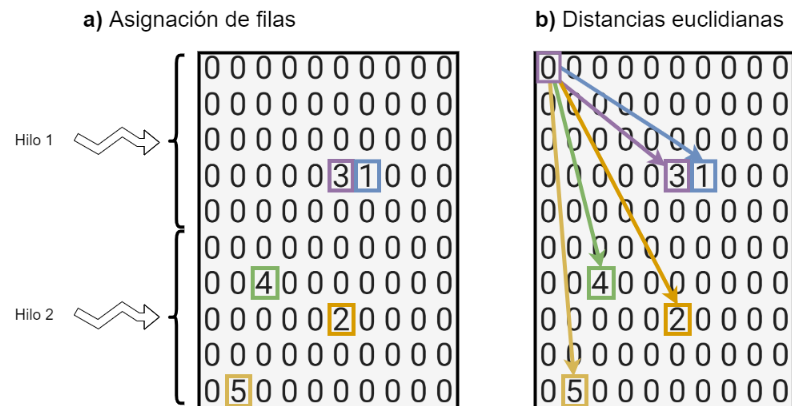


Figura 7.3: Ejemplo de asignación de filas a los hilos y cálculo de distancias euclidianas.

La implementación en GLG-E se detalla a continuación considerando el ejemplo anterior (matriz de 10 x 10 y 5 regiones) a fin de visualizar el trabajo con GLG-E. Posteriormente, se escala el tamaño del problema para comparar los tiempos de ejecución.

En el primer ícono de *Cabecera* se selecciona en especial la biblioteca *math_functions*, la cual permite calcular potencias y raíces cuadradas. Otras bibliotecas utilizadas se presentan en la Figura 7.4.

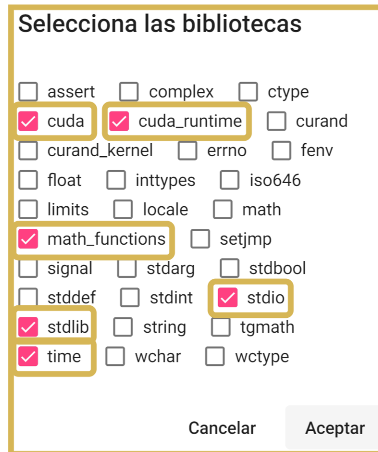


Figura 7.4: Selección de bibliotecas para el programa *diagramas de Voronoi*.

En el ícono de *Constantes* (Figura 7.5a), se establecen las constantes necesarias para el programa de generación de diagramas Voronoi. En este caso, se define el número de filas (REN) y columnas (COL) para crear una matriz de tamaño 10 x 10. Además, se establece el número de regiones deseado, que para este ejemplo es de 5.

Para las variables de comunicación, se crean dos arreglos (pX y pY) que contienen las posiciones i y j de los generadores. Estos arreglos tienen una longitud de 5, correspondiente al número de regiones. También se crea un arreglo unidimensional con 100 elementos (*m_aplanada*) para el envío de la matriz generada (Figura 7.5b). Posteriormente se explicará el proceso de aplanamiento de la matriz para su envío a la GPU.

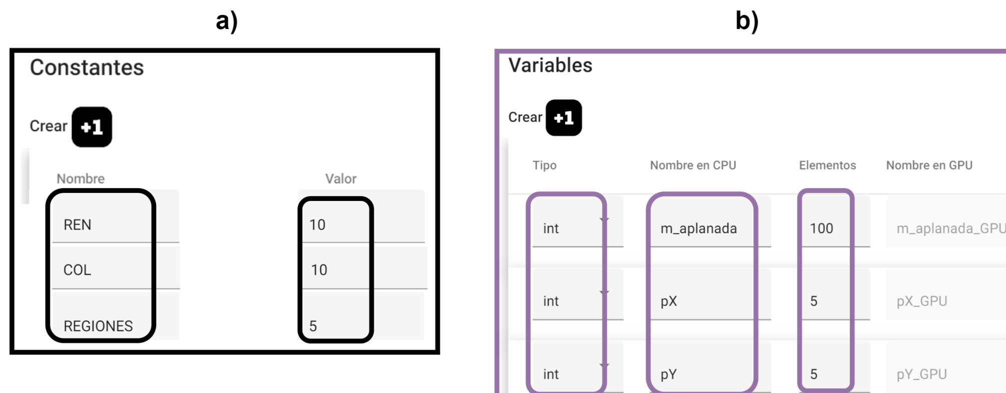


Figura 7.5: Propiedades de los íconos de *Constantes* y *Variables de comunicación* para la creación del programa *diagramas de Voronoi.*

En el ícono *CPU de entrada*, se realiza la preparación de los datos que serán procesados por el kernel. En primer lugar, se generan de manera aleatoria las posiciones i y j de los generadores. Estas posiciones se ajustan dentro del rango de la matriz de tamaño 10 x 10. Las posiciones generadas se almacenan en las variables de comunicación pX y pY.

Luego, se crea una matriz de tamaño 10 x 10 y se asigna el valor cero a cada elemento de la matriz. A continuación, se colocan los generadores en la matriz utilizando las posiciones de pX y pY. Cada generador se incluye en la matriz con un valor del 1 al 5, como se muestra en la Figura 7.6.

Esta etapa de preparación de datos es necesaria para establecer las condiciones iniciales del programa y generar la matriz que servirá de base para la generación de los diagramas Voronoi.

Figura 7.6: Asignación de generadores dentro de la matriz.

El proceso de envío de datos hacia la GPU requiere que la matriz sea convertida a un arreglo unidimensional, ya que solo se permite el envío de datos en esta forma a la GPU. Para lograr esto, los elementos de la matriz se almacenan en secuencia en un arreglo.

Es importante destacar que, aunque se realiza la conversión a un arreglo unidimensional, es posible volver a convertir el arreglo en una matriz dentro de la GPU, manteniendo así un nivel de abstracción más comprensible. Sin embargo, en este caso particular se mantiene la representación en forma unidimensional para facilitar el proceso de envío de datos nuevamente a la CPU.

De esta manera se envían 3 variables a la GPU: la matriz en su forma unidimensional, las posiciones i y j de los generadores en pX y pY (nótese la selección de estas variables en las propiedades de los íconos de *CPU de entrada* y *salida* en la Figura 7.7).

Una vez que los datos han sido procesados en el kernel de la GPU, se envía de regreso a la CPU la matriz en su forma unidimensional. Esto se logra mediante la transferencia de datos desde la GPU a la CPU. Posteriormente, la matriz resultante en su forma unidimensional se utiliza en el ícono de *CPU de salida* para imprimir la matriz generada.

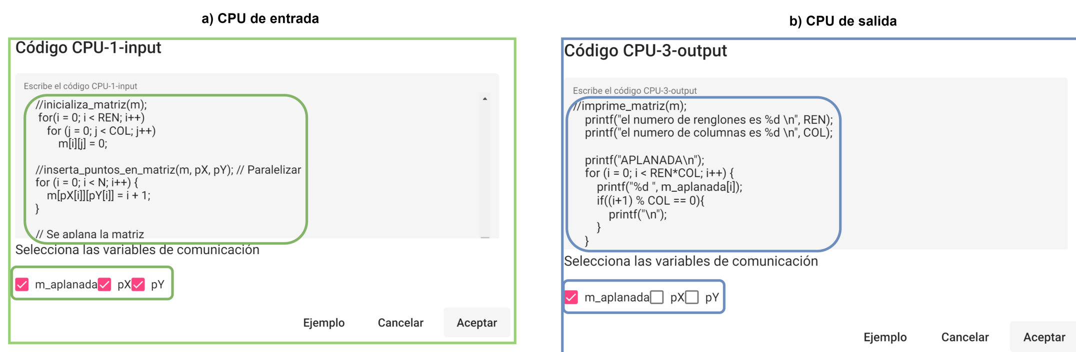


Figura 7.7: Llenado y selección de propiedades de los íconos de *CPU de entrada* y *salida* para la creación del programa *diagramas de Voronoi*.

La generación del diagrama de Voronoi se lleva a cabo en la GPU utilizando un algoritmo que determina a qué generador pertenece cada elemento de la matriz. Este proceso se realiza mediante el cálculo de las distancias euclidianas entre el elemento y cada uno de los puntos generadores. Como se explicó ante-

riormente, cada hilo asociado a un conjunto de filas, asigna para cada elemento la región correspondiente en función de su cercanía a los generadores. De esta manera, se logra la generación del diagrama de Voronoi de manera eficiente y paralela, aprovechando el poder de procesamiento masivo que ofrece esta unidad de procesamiento.

Código GPU-2

N° Hilos 100

b) Código de la GPU

Escribe el código GPU-2

```
int i, j, k;
int pos_min;
double distancia;
double distancia_min;

// CADA HILO TIENE UN BLOQUE DE FILAS
int bloque = REN / (HILOS*BLOQUE);
int resto = REN % (HILOS*BLOQUE);
int inicio = idglobal * bloque;
int fin = inicio + bloque;

for (i = inicio; i < fin; i++){
  for (j = 0; j < COL; j++) {
    if (m_aplanada[i*COL+j] == 0) // calcular mas cercano
    {
      pos_min = 0;
      distancia_min = sqrtf(powf((pX[pos_min]) - i, 2) + powf((pY[pos_min]) - j, 2));
      for (k = 1; k < N; k++) {
        distancia = sqrtf(powf((pX[k]) - i, 2) + powf((pY[k]) - j, 2));
        if (distancia < distancia_min) {
          pos_min = k;
          distancia_min = sqrtf(powf((pX[k]) - i, 2) + powf((pY[k]) - j, 2));
        }
      } // fin del for
      m_aplanada[i*COL+j] = pos_min + 1;
```

Figura 7.8: Llenado de las propiedades del ícono de *GPU* para el programa *diagramas de Voronoi*.

A continuación, se presenta en la Figura 7.9 un fragmento del código generado (de manera automática) a partir de las traducciones de las propiedades de los íconos en GLG-E. Este fragmento de código es solo una representación parcial del programa completo, pero permite visualizar la estructura y las operaciones involucradas en la generación del diagrama de Voronoi.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <math.h>
#include <math_functions.h>

#define REN 10
#define COL 10
#define REGIONES 5

__global__ void CPU_1_input(int *m_aplanada, int *pX, int *pY) {
    int idHiloBloque= ((threadIdx.x * blockDim.y) + threadIdx.y) + (blockDim.x*blockDim.y)*threadIdx.z;
    int idBloqueGrid =  ((blockIdx.x * gridDim.y) + blockIdx.y) + (gridDim.x*gridDim.y)*blockIdx.z;
    int idglobal = idBloqueGrid*(blockDim.x*blockDim.y*blockDim.z) + idHiloBloque;
    int i, j, k;
    int pos_min;
    double distancia;
    double distancia_min;

    // CADA HILO TIENE UN BLOQUE DE FILAS
    int bloque = REN / (HILOS*BLOQUE);
    int resto = REN % (HILOS*BLOQUE);
    int inicio = idglobal * bloque;
    int fin = inicio + bloque;

    for (i = inicio; i < fin; i++){
        for (j = 0; j < COL; j++) {
            if (m_aplanada[i*COL+j] == 0){
                pos_min = 0;
                distancia_min = sqrtf(powf((pX[pos_min]) - i, 2) + powf((pY[pos_min]) - j, 2));
                for (k = 1; k < N; k++) {
                .
                .
                .
```

Figura 7.9: Fragmento de código generado por GLG-E para el programa *diagramas de Voronoi*.

El resultado final del programa es una matriz que representa visualmente el diagrama de Voronoi. Cada región del diagrama se distingue por un color o un valor específico asignado a los elementos de la matriz (Figura 7.10). Esta representación permite visualizar de forma clara y concisa las diferentes regiones generadas a partir de los generadores establecidos.

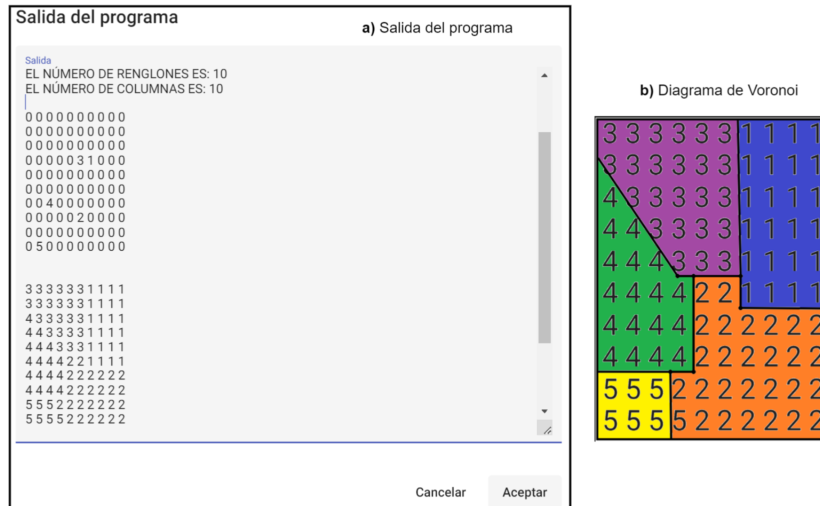


Figura 7.10: Salida del programa *diagramas de Voronoi*.

7.2.2.2. Área bajo la curva

Para este caso de prueba, se plantea la aproximación del área bajo la curva de la función $f(x) = x^2 + 1$ en el intervalo $[1, 3]$ utilizando el método de las sumas de Riemann. El número de particiones (rectángulos) se establece en 50 millones para lograr una aproximación precisa del área.

Además, se define el intervalo de la función asignando los valores de $a = 1$ y $b = 3$. Luego, se calcula el tamaño de cada subintervalo, también conocido como Δx, utilizando la fórmula $\Delta x = \frac{b-a}{\text{particiones}}$.

Una estrategia para calcular de forma eficiente y rápida las evaluaciones de puntos en paralelo, es particionar el intervalo $[a, b]$ entre el número de hilos para calcular la suma de las áreas parciales de los rectángulos (Figura 7.11).

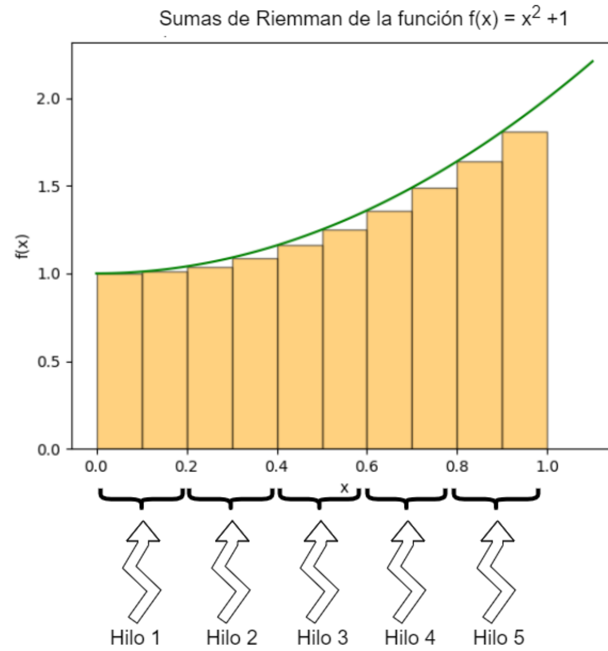


Figura 7.11: Ejemplo de la asignación de sub-intervalos a los hilos.

Una vez que cada hilo ha finalizado el cálculo de su área parcial, se realiza la suma total de todas las áreas parciales en la CPU. Esto se hace para obtener la aproximación del área total bajo la curva.

En la Figura 7.12 a) y b), muestran tanto la configuración de las propiedades del ícono de *Cabecera*, así como del ícono de *Constantes* con los valores previamente mencionados.

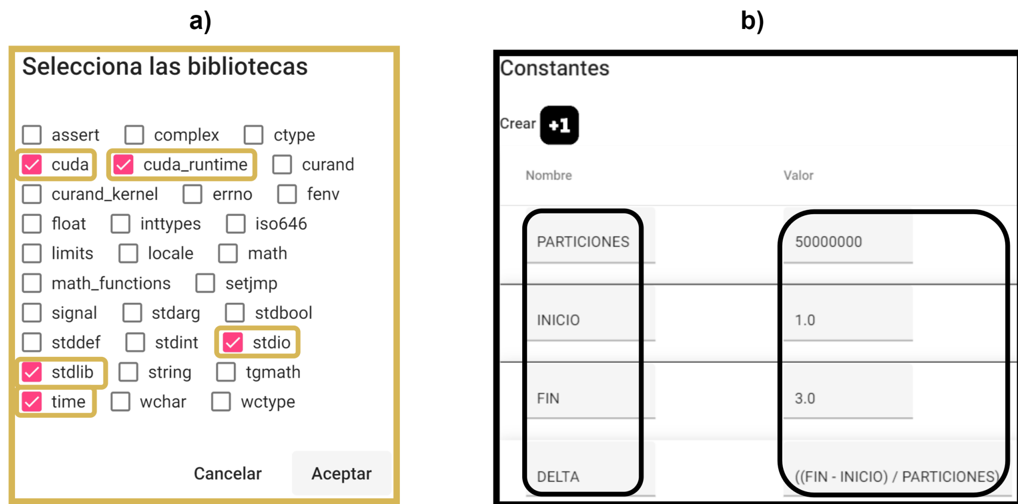


Figura 7.12: Propiedades de los íconos de *Cabecera* y *Constantes* para la creación del programa *área bajo la curva*.

En el ícono de *Variables de comunicación* (Figura 7.13a) se crea un arreglo para almacenar el cálculo del área del subintervalo correspondiente a cada hilo. En este caso, cada hilo se encarga de calcular el área de un subintervalo específico.

En el ícono de *CPU de entrada* (Figura 7.13b), no es necesario escribir código adicional, ya que el kernel se encargará de realizar el cálculo del área utilizando los datos proporcionados. En esta etapa, solo se selecciona la variable que se enviará a la GPU para su procesamiento.

Figura 7.13: Propiedades de los íconos de *Variables de comunicación* y *CPU de entrada* para la creación del programa *área bajo la curva*.

En el ícono de *GPU* (Figura 7.14a), cada uno de los hilos obtiene el subintervalo que le corresponde para calcular el área parcial de ese rango. Una vez hecho

esto, se procede a calcular la suma de las áreas de los rectángulos. Finalmente cada hilo guarda el resultado en un arreglo compartido, para posteriormente regresar los resultados a la CPU.

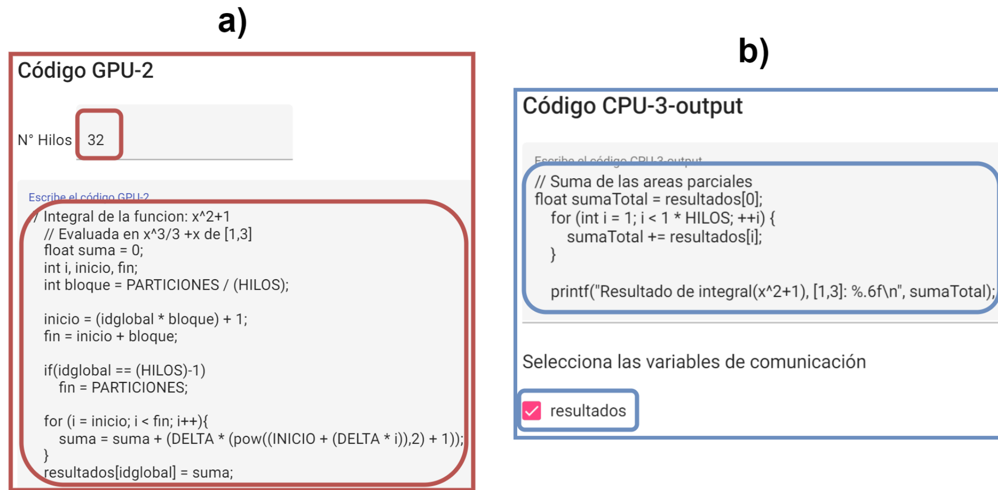


Figura 7.14: Propiedades de los íconos de *GPU* y *CPU de salida* para la creación del programa *área bajo la curva.*

El código correspondiente de la *CPU de salida* (Figura 7.14b) se encarga de realizar la suma de las áreas parciales que son guardadas en el arreglo de resultados. Finalmente se muestra el resultado de la suma total de las áreas para dar una aproximación al valor real. Nuevamente, una parcialidad del código generado por GLG, así como la salida se presentan en la Figura 7.15 a) y b), respectivamente.

a)

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <time.h>

#define PARTICIONES 50000000
#define INICIO 1.0
#define FIN 3.0
#define DELTA ((FIN - INICIO) / PARTICIONES)
#define HILOS 32

__global__ void CPU_1_input(float* resultados) {
    int idHiloBloque= ((threadIdx.x * blockDim.y) + threadIdx.y) + (blockDim.x*blockDim.y)*threadIdx.z;
    int idBloqueGrid =  ((blockIdx.x * gridDim.y) + blockIdx.y) + (gridDim.x*gridDim.y)*blockIdx.z;
    int idglobal = idBloqueGrid*(blockDim.x*blockDim.y*blockDim.z) + idHiloBloque;
    // Integral de la funcion: x^2+1
    // Evaluada en x^3/3 +x de [1,3]
    float suma = 0;
    int i, inicio, fin;
    int bloque = PARTICIONES / (HILOS);

    inicio = (idglobal * bloque) + 1;
    fin = inicio + bloque;

    if(idglobal == (HILOS)-1)
        fin = PARTICIONES;

    for (i = inicio; i < fin; i++){
        suma = suma + (DELTA * (pow((INICIO + (DELTA * i)),2) + 1));
    }
    resultados[idglobal] = suma;
}

int main(){
```

Salida del programa

b)

```
Salida
Resultado de integral(x^2+1), [1,3]: 10.648619

    *****************************************************************
    * TIEMPO DE EJECUCIÓN DEL PROGRAMA: 860 ms, 0.8600 s
    *****************************************************************
```

Figura 7.15: Parcialidad del código generado y salida de la ejecución del programa *área bajo la curva*.

7.2.3. Evaluación de desempeño

En esta sección se muestra la comparación de desempeño del código generado en GLG respecto a la solución directamente en CUDA. Para ambas gráficas, el eje vertical representa el tiempo de ejecución y el eje horizontal el número de hilos utilizados en la tarjeta gráfica. Se realizaron 100 ejecuciones tanto del código generado por GLG como de la versión textual escrita de forma tradicional. La plataforma de ejecución utilizada es un sistema operativo Linux CentOS 7.9, con *CUDA Driver Version 11.6* y una tarjeta gráfica NVIDIA *GeForce GTX 1080 Ti*.

7.2.3.1. Diagrama Voronoi

En la Figura 7.16 se muestra el tiempo de ejecución del programa *diagramas de Voronoi* utilizando una matriz de dimensiones 17920 x 17920. Esta matriz se eligió como un múltiplo del máximo número de hilos de la tarjeta gráfica. Se establece el número de generadores (regiones) en 10 y se escala el número de hilos en 256, 512, 1024, 2048, 2560, 3072 y 3584. Para cada asignación de hilos, se ejecutó el programa 100 veces para promediar los tiempos y lograr una tendencia. El eje X representa el número de hilos utilizados y el eje Y corresponde al tiempo de ejecución en milisegundos.

Como se puede observar, la tendencia de la curva disminuye debido a que a medida que se utilizan más hilos para abordar el problema, la carga de trabajo se distribuye entre los hilos, lo que reduce el tiempo de ejecución. Además, observamos que el costo en tiempo del código generado no afecta en general el desempeño de la solución debido a que las líneas de la gráfica están superpuestas.

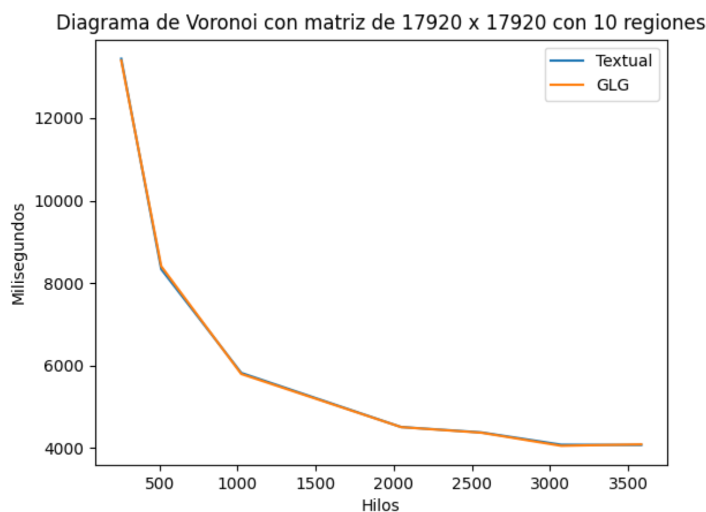


Figura 7.16: Tiempos del programa *diagramas de Voronoi*.

7.2.3.2. Área bajo la curva

De manera similar al caso de prueba *Diagramas de Voronoi*, se escala el tamaño de hilos en 256, 512, 1024, 2048, 2560, 3072 y 3584 y se realizaron los

promedios de 100 ejecuciones del programa textual y de 100 ejecuciones del programa generado por GLG para lograr una tendencia en los resultados.

Los tiempos obtenidos al ejecutar el programa *área bajo la curva* se muestran en la Figura 7.17. Podemos observar que el rendimiento de ambas versiones es muy similar. Notar que como es de esperarse, los tiempos de respuesta se reducen al aumentar el número de hilos. Además, al igual que en el caso anterior las líneas de la gráfica están superpuestas por lo que se deduce igualmente que el código generado no afecta en el tiempo de ejecución.

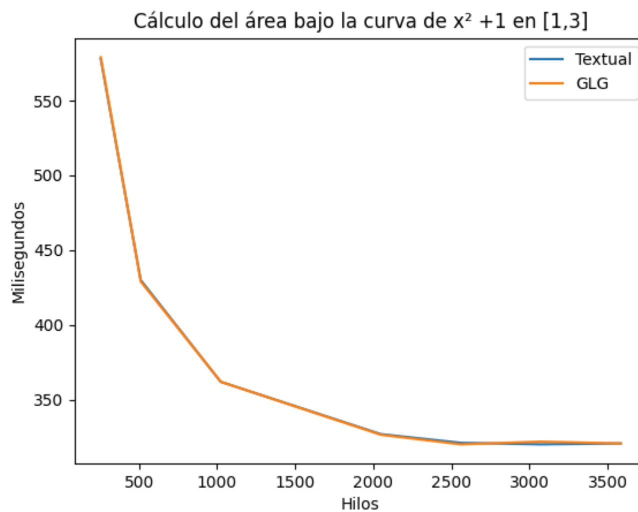


Figura 7.17: Tiempos del programa *área bajo la curva*.

Como se muestra en ambos programas, los resultados de la versión textual y la versión en GLG tienen tiempos muy similares. Esto demuestra que GLG es capaz de generar código eficiente y optimizado, logrando un rendimiento comparable al código escrito en lenguaje textual. Esto es una ventaja significativa, ya que GLG permite a los programadores aprovechar los beneficios de la programación visual sin comprometer el rendimiento de sus aplicaciones.

Resumen

En los casos de prueba presentados se exploraron diferentes programas paralelizables utilizando GLG-E. Estos programas incluyen la generación de diagramas de Voronoi y el cálculo del área bajo la curva mediante las sumas de Riemann.

Cada caso de prueba demostró la capacidad de GLG-E para generar código CUDA de forma automática, destacando la eficiencia y versatilidad de GLG-E en la programación paralela.

Capítulo 8

Conclusiones y trabajo futuro

En este trabajo, se propuso un lenguaje visual innovador para la programación en tarjetas gráficas NVIDIA. El lenguaje se compone de un conjunto de íconos que representan diferentes elementos de un código de CUDA, como bibliotecas, constantes, variables de comunicación y el procesamiento en la CPU y GPU. Además, se ha definido una gramática que permite construir programas de manera lógicamente estructurada, lo que mejora la aplicabilidad del lenguaje en diversos escenarios.

Además de la definición del lenguaje, se desarrolló un entorno web que brinda soporte para programar en el lenguaje propuesto. Este entorno hace uso de una API-Rest, lo cual proporciona una serie de servicios adicionales, como la verificación de conexión, obtención de características de la tarjeta gráfica, monitoreo de procesos y descarga de códigos de ejemplo. Esta arquitectura contribuyó a la portabilidad de GLG-E, permitiendo su ejecución en diferentes plataformas y dispositivos.

GLG y GLG-E ofrecen varias ventajas en comparación con los lenguajes textuales tradicionales, como el uso de íconos y flujos de trabajo para facilitar la comprensión y visualización de los programas. La generación automática de código y la posibilidad de realizar modificaciones manuales brindan flexibilidad y adaptabilidad al lenguaje, lo que resulta en un equilibrio entre la simplicidad de programación visual y la complejidad de la programación textual.

Se presentó la utilidad y efectividad de GLG a través de la implementación de programas de prueba y la comparación con lenguajes textuales. Además, con-

sideramos que la arquitectura web garantiza la portabilidad y accesibilidad del entorno, lo que lo convierte en una herramienta versátil para la comunidad de programadores interesados en el procesamiento paralelo en tarjetas gráficas.

Como trabajo a futuro, se plantean las siguientes líneas de investigación y mejora:

- **Identificación de nuevos iconos:** Identificar y desarrollar nuevos iconos que puedan facilitar aún más la programación en tarjetas gráficas. Estos nuevos iconos podrían abarcar funcionalidades adicionales o tareas específicas que no están cubiertas actualmente como estructuras de repetición (*for* y *while*) y condicionales (*if-else*).

- **Mejora de la interfaz de usuario y experiencia de usuario**: Realizar mejoras en la interfaz de usuario y en la experiencia de usuario de la aplicación web. Lo anterior incluiría ajustes en la disposición de los iconos, la organización de los flujos de trabajo y la implementación de funcionalidades intuitivas que faciliten la interacción con la herramienta.

- **Soporte para datos n-dimensionales:** Trabajar en la incorporación de soporte para el manejo de datos n-dimensionales en el lenguaje visual. Esto ofrecería a los programadores trabajar con conjuntos de datos más complejos.

- **Optimización del cálculo de hilos basado en el *warp*:** Investigar y desarrollar técnicas para optimizar el cálculo de hilos en función del *warp* en tarjetas gráficas. Esto permitiría aprovechar al máximo los recursos de la GPU y mejorar el rendimiento de los programas generados por el lenguaje visual.

- **Implementación de funcionalidad de guardado y carga de archivos propios de GLG:** Trabajar en la implementación de funcionalidades que permitan guardar y cargar archivos en un formato propio de GLG. Esto facilitaría el intercambio de programas entre diferentes usuarios y la reutilización de código.

- **Creación de métodos para la GPU y CPU:** Investigar y desarrollar una forma de definir y utilizar métodos que puedan ser ejecutados tanto en la GPU como en la CPU. Esto permitiría una mayor flexibilidad al diseñar programas y aprovechar las capacidades de ambos procesadores.

Otro trabajo a futuro sería el realizar encuestas y pruebas de usabilidad, para validar el diseño de la interfaz y la experiencia de usuario de la aplicación. Esto permitiría recopilar comentarios y sugerencias de los usuarios para realizar ajustes y mejoras adicionales.

Bibliografía

[ADM22] Nisha Agrawal, Abhishek Das, and Manish Modani. Scalability analysis of weather research forecast model on nvidia ampere based dense gpu cluster. In *2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS)*, pages 1–6, 2022. 1

[BHD+95] James Browne, Syed Hyder, Jack Dongarra, Keith Moore, and Peter Newton. Visual programming and debugging for parallel computing. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 3:75 – 83, 02 1995. 27

[Dav19] David Aarón Velasco Romero. *Desarrollo de varios métodos numéricos sobre GPUs y su aplicación en entornos astrofísicos.* Tesis de doctorado, Universidad Autónoma del Estado de Morelos, Mayo 2019. 61

[DKH17] F. Drewes, H.J. Kreowski, and A. Habel. *Hyperedge Replacement Graph Grammars*, chapter 2, pages 95–162. Printed in Singapore, 1917. 19

[DMB16] H. V. Deepika, N. N. Mangala, and Sarat Chandra Babu. Automatic program generation for heterogeneous architectures. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 102–109, 2016. 2, 14, 16, 18

[Doz01] Gabor Dozsa. *Visual Programming to Support Parallel Program Design*, page 17–44. Nova Science Publishers, Inc., USA, 2001. 11, 12, 13

[EM96] G. Armando Eduardo and N. Marcelo. Experiencias en la enseñanza de programación paralela. *Congreso Argentino de Ciencias de la Computación*, pages 549–558, 1996. 2

[ESW17] Martin Erwig, Karl Smeltzer, and Xiangyu Wang. What is a visual language? *Journal of Visual Languages & Computing*, 38:9–17, 2017. SI:In honor of Prof SK Chang. 2, 9

[Fab19] José Luis Quiroz Fabián. *Lenguaje Visual para el Desarrollo de Programas Paralelos*. PhD thesis, Universidad Autónoma Metropolitana, 2019. 6, 12

[GOY89] Michael T. Goodrich, Colm ODunlaing, and Chee K. Yap. Constructing the voronoi diagram of a set of line segments in parallel. In F. Dehne, J. R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, pages 12–23, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. 62

[HLPT19] Yuan-Shin Hwang, Hsih-Hsin Lin, Shen-Hung Pai, and Chia-Heng Tu. Gpublocks: Gui programming tool for cuda and opencl. *Journal of Signal Processing Systems*, 91, 03 2019. 2, 14, 17, 18

[LTH15] Hsih-Hsin Lin, Chia-Heng Tu, and Yuan-Shin Hwang. Cudablock: A gui programming tool for cuda. In *2015 44th International Conference on Parallel Processing Workshops*, pages 37–42, 2015. 2, 14, 17, 18

[Mar15] A. Marcelo. Enseñanza de programación paralela y distribuida en las carreras de grado de computación. *Congreso sobre Tecnología en Educación & Educación en Tecnología (TE & ET) (Corrientes, 2015)*, 2015. 2

[ME09] Bryan McDonnel and Niklas Elmqvist. Towards utilizing gpus in information visualization: A model and implementation of image-space operations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1105–1112, 2009. 1

[MIT09] MIT. App inventor. `https://appinventor.mit.edu/`, 2009. 10

[NMRALG+20] Aurelio Nicolás Mata, Graciela Román Alonso, Gabriel López Garza, José Rafael Godinez Fernández, Miguel Alfonso Castro García, and Norma Pilar Castellanos Ábrego. Parallel simulation of the synchronization of heterogeneous cells in the sinoatrial node. *Concurrency and Computation: Practice and Experience*, 32(10):e5317, 2020. e5317 cpe.5317. 1

[Red22] Red Hat. What are application programming interfaces?, Last update: 2022. 2

[Rod15] Ulises Martínez Rodríguez. Aplicación de la geometría computacional en la reconstrucción 3d basada en diagramas de voronoi, 2015. 62

[Roq08] Ricarose Vallarta Roque. Openblocks : an extendable framework for graphical block programming systems. 05 2008. 14, 27

[WM17] Akiyoshi Wakatani and Toshiyuki Maeda. Web applications for learning cuda programming. In *2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA)*, pages 1–5, 2017. 2

[ZB15] Simon Portegies Zwart and Jeroen Bédorf. Using gpus to enable simulation with computational gravitational dynamics in astrophysics. *Computer*, 48(11):50–58, 2015. 1