Instituto Politécnico Nacional

“Escuela Superior de Cómputo”

Integrantes:

- Diaz Ruiz Israel
- Ignacio Cortés Atzin Maxela
- Ríos Rivera Fernanda Anahí

Unidad de aprendizaje: Cómputo paralelo

Profesor: Luis Alberto Ibáñez Zamora

Grupo: 6BM2

“Reporte de práctica 8”

Evaluating element-wise expressions with PyCUDA

índice

Índice de figuras

Resumen Práctico

Este proyecto tiene como objetivo demostrar cómo utilizar la clase pycuda.elementwise.ElementwiseKernel para realizar una combinación lineal de vectores directamente en la GPU, sin necesidad de escribir manualmente un kernel CUDA. Se presenta un caso práctico que ilustra cómo se pueden aprovechar las capacidades paralelas de la GPU mediante una interfaz sencilla y eficiente ofrecida por PyCUDA, permitiendo operaciones elemento a elemento sobre arreglos almacenados en memoria de dispositivo. Este enfoque permite simplificar el desarrollo de algoritmos numéricos intensivos, reduciendo la complejidad asociada a la programación en CUDA puro y facilitando su integración con Python.

Introducción

En aplicaciones científicas y de ingeniería, es común encontrar operaciones matemáticas que deben aplicarse de forma independiente a cada elemento de un arreglo. Estas operaciones son naturalmente paralelizables, lo cual las convierte en candidatas ideales para ejecutarse en GPU. Sin embargo, tradicionalmente esto requería escribir código CUDA desde cero, lo cual implica un alto costo de desarrollo y curva de aprendizaje.

PyCUDA ofrece una alternativa accesible mediante ElementwiseKernel, una herramienta que encapsula la complejidad del lanzamiento de hilos y la gestión de índices, permitiendo definir operaciones matemáticas en notación C-like dentro del entorno Python. Esto posibilita acelerar operaciones numéricas comunes —como sumas, multiplicaciones escalares, funciones trigonométricas o combinaciones lineales— sin abandonar la simplicidad y productividad del lenguaje Python.

El presente trabajo explora el uso de esta herramienta para implementar una combinación lineal de dos vectores (c[i] = a * x[i] + b * y[i]) en GPU, evaluando su rendimiento y precisión frente a una implementación equivalente en CPU usando NumPy.

Metodología

El desarrollo del proyecto se llevó a cabo mediante una metodología estructurada que permitió integrar de forma eficiente la programación en GPU con el entorno científico de Python. Inicialmente, se generaron dos vectores aleatorios de números en punto flotante utilizando la función `curand.rand` de PyCUDA, lo cual evitó la necesidad de generar los datos en CPU y luego transferirlos a la GPU, optimizando así el flujo de trabajo. Estos vectores fueron almacenados directamente en memoria de dispositivo mediante el uso de `pycuda.gpuarray.GPUArray`, facilitando su acceso rápido durante las operaciones posteriores. Posteriormente, se definieron los coeficientes escalares que participan en la combinación lineal, los cuales se pasan como argumentos constantes al kernel que opera sobre los datos. La operación matemática central —una combinación lineal de la forma

`c[i] =a*x[i]+b*y[i]` fue implementada mediante el uso de `pycuda.elementwise.ElementwiseKernel`, una herramienta que encapsula la complejidad de la programación CUDA tradicional, permitiendo definir operaciones elemento a elemento de forma intuitiva, sin necesidad de escribir manualmente bloques de código para el lanzamiento de hilos ni la gestión de índices. Una vez creado el kernel, se preparó un arreglo de salida en GPU del mismo tamaño que los vectores de entrada, asegurando espacio suficiente para almacenar el resultado. A continuación, se invocó el kernel pasando los coeficientes escalares y los punteros a los arreglos de entrada y salida, ejecutándose automáticamente en paralelo sobre todos los elementos de los vectores. Finalmente, el resultado fue recuperado desde la GPU hacia la CPU para realizar una validación numérica cruzándolo contra el cálculo equivalente realizado con NumPy, empleando funciones como `numpy.linalg.norm` para medir la diferencia absoluta entre ambos resultados y verificar que esta se encontrara dentro de una tolerancia aceptable (`1e-5`), típica en operaciones de punto flotante en entornos paralelos. Este proceso no solo asegura la funcionalidad del kernel, sino también su precisión y confiabilidad en aplicaciones científicas reales. Todo el flujo fue diseñado siguiendo buenas prácticas de programación híbrida CPU-GPU, manteniendo claridad en el código, separación de responsabilidades y manejo seguro de recursos computacionales.

Implementación del código

Instrucciones

Este proyecto tiene como finalidad explorar el uso de `pycuda.elementwise.ElementwiseKernel` para realizar operaciones matemáticas elemento a elemento en GPU, sin necesidad de escribir kernels CUDA manuales. Se implementa una combinación lineal de vectores que utiliza escalares y arreglos en memoria de GPU, validando su resultado contra un cálculo equivalente en CPU con NumPy. Además, se analiza la precisión numérica considerando el error inherente al punto flotante en cómputo paralelo, se comparan tiempos de ejecución entre CPU y GPU, y se promueven buenas prácticas de programación híbrida. También se explora la extensibilidad del kernel para incluir funciones más complejas, todo ello vinculado con los conceptos básicos de la arquitectura CUDA, como la asignación de hilos a elementos de un vector y la ejecución paralela masiva.
Una operación element-wise (elemento a elemento) es una operación que se aplica individualmente a cada posición de un arreglo.

Ejemplos comunes incluyen:

- Suma: $c[i] = a[i] + b[i]$
- Producto: $c[i] = a[i] \times b[i]$
- Escalado: $a[i] = k \cdot a[i]$
- Función matemática: $b[i] = \sin(a[i])$

pycuda.elementwise.ElementwiseKernel

ElementwiseKernel permite crear operaciones GPU personalizadas para cada elemento de un vector sin escribir manualmente un kernel CUDA completo.

Análisis y Discusión

Ilustración 1 Se instala cuda en Google colab

```python
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
import numpy as np
from pycuda.curandom import rand as curand
from pycuda.elementwise import ElementwiseKernel
import numpy.linalg as la


# 1. Crear dos vectores de entrada aleatorios en GPU
input_vector_a = curand((50,))
input_vector_b = curand((50,))
# 2. Coeficientes escalares para la combinación lineal
mult_coefficient_a = 2.0
mult_coefficient_b = 5.0

# 3. Definir el kernel de combinación lineal: c[i] = a * x[i] + b * y[i]
linear_combination = ElementwiseKernel(
"float a, float *x, float b, float *y, float *c",
"c[i] = a * x[i] + b * y[i];",
"linear_combination"
)


# 4. Vector de salida en GPU
linear_combination_result = gpuarray.empty_like(input_vector_a)

# 5. Ejecutar el kernel
linear_combination(
mult_coefficient_a, input_vector_a,
mult_coefficient_b, input_vector_b,
linear_combination_result
)
```

Ilustración 2 Implementación del código

```python
# 6. Imprimir resultados
print(" INPUT VECTOR A:\n", input_vector_a)
print(" INPUT VECTOR B:\n", input_vector_b)
print(" RESULTING VECTOR C (GPU):\n", linear_combination_result)

# 7. Validación: comparar con NumPy (CPU)
expected_result = mult_coefficient_a * input_vector_a + mult_coefficient_b * input_vector_b
difference = linear_combination_result - expected_result
print(" DIFFERENCE (C - (a*A + b*B)):\n", difference)

assert la.norm(difference.get()) < 1e-5, "Error: resultado fuera de tolerancia"
print(" Validación correcta: El resultado es numéricamente equivalente.")
```

Ilustración 3 Implementación del códigp

Este código utiliza PyCUDA para realizar una combinación lineal de dos vectores directamente en la GPU, sin necesidad de escribir un kernel CUDA manual. Primero, se generan dos vectores aleatorios en la GPU usando curand, y se definen dos coeficientes escalares. Luego, se utiliza ElementwiseKernel para definir una operación paralela que calcula $c[i]=a \cdot x[i]+b \cdot y[i]c[i] = a * x[i] + b * y[i]$, donde cada hilo CUDA procesa un elemento del vector. El resultado se almacena en un arreglo de salida también en GPU.

Finalmente, se compara este resultado con el cálculo equivalente hecho en CPU usando NumPy, y se verifica que la diferencia numérica sea menor a una tolerancia predefinida (1e-5). Si la validación es exitosa, se confirma que la operación en GPU es correcta y equivalente

en precisión, demostrando que ElementwiseKernel es una herramienta eficiente y práctica para realizar operaciones matemáticas sencillas en paralelo.

```
INPUT VECTOR A:
[0.28514048 0.8086949  0.02722696 0.411991   0.6622893  0.47768903
0.33592314 0.5840904  0.21026868 0.5325989  0.11485961 0.11215468
0.54633564 0.23533958 0.374626   0.74959385 0.18406704 0.4422752
0.8937354  0.49610016 0.57345563 0.4645909  0.23851304 0.06214578
0.9690773  0.4187021  0.7547976  0.2296456  0.05650645 0.9019539
0.69223374 0.7319228  0.58873343 0.07183412 0.628524   0.8623117
0.21442437 0.9432447  0.76435596 0.5123143  0.18388045 0.00900882
0.4478213  0.64425164 0.20903389 0.7116797  0.53257424 0.36030287
0.19848508 0.6887586 ]
INPUT VECTOR B:
[0.34579507 0.91793334 0.30946156 0.40048942 0.8437291  0.22018342
0.7656651  0.03734918 0.8624585  0.6424892  0.918996   0.1848841
0.84359026 0.85252506 0.99673414 0.4154753  0.7319554  0.81351846
0.7417453  0.2652009  0.2353548  0.5116927  0.6233376  0.07267116
0.8331671  0.88907135 0.7040162  0.29364124 0.95764077 0.8803491
0.7747229  0.25513232 0.56139535 0.75177175 0.02386324 0.15287742
0.7111007  0.14106324 0.36066812 0.3991603  0.33854553 0.8152324
0.11473598 0.58430946 0.89961356 0.28403413 0.2023961  0.9045145
0.39903066 0.4777228 ]
RESULTING VECTOR C (GPU):
[2.2992563  6.2070565  1.6017618  2.8264291  5.5432243  2.0562952
4.5001717  1.3549267  4.73283    4.2776437  4.824699   1.1487298
5.3106227  4.7333045  5.7329226  3.5765643  4.027911   4.9521427
5.496197   2.3182049  2.3236852  3.4876454  3.5937142  0.48764735
6.10399    5.282761   5.0296764  1.9274974  4.9012165  6.205653
5.258082   2.7395072  3.9844437  3.9025269  1.3763642  2.4890106
3.984352   2.5918055  3.3320527  3.02043    2.0604887  4.0941796
1.4693224  4.2100506  4.916136   2.8435302  2.077129   5.2431784
2.3921235  3.7661312 ]
DIFFERENCE (C - (a*A + b*B)):
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0.]
Validación correcta: El resultado es numéricamente equivalente.
```

Ilustración 4 Resultados

La imagen muestra la ejecución exitosa de un programa en PyCUDA que calcula una combinación lineal de dos vectores directamente en la GPU mediante `ElementwiseKernel`. Se observa que los vectores de entrada `A` y `B` contienen 50 valores aleatorios generados en la memoria de la GPU. Los coeficientes escalares definidos son `a = 2.0` y `b = 5.0`, lo que implica que el vector resultante `C` fue calculado como $C = 2A + 5B$, elemento a elemento, usando cómputo paralelo en GPU.

El resultado de esta operación se presenta en el vector `C`, con valores consistentes con la combinación lineal esperada. A continuación, se realiza una validación comparando este resultado con la misma operación efectuada en CPU mediante NumPy, y se muestra la diferencia entre ambos vectores. La diferencia resultante es un vector lleno de ceros, indicando que no hay discrepancias numéricas significativas.

Finalmente, se confirma con el mensaje que el resultado es numéricamente equivalente, lo que valida que el cálculo realizado en GPU es preciso y confiable, dentro de la tolerancia aceptable para operaciones en punto flotante. Esto demuestra la utilidad de `ElementwiseKernel` para acelerar operaciones vectoriales simples sin comprometer la exactitud del resultado.

Código

https://colab.research.google.com/drive/10zGDqibxQXaoZC4hqKYfsYGknn0MHeu-?usp=sharing

Conclusiones

La ejecución del código demuestra que el uso de pycuda.elementwise.ElementwiseKernel es una forma eficiente, precisa y sencilla de realizar operaciones matemáticas vectoriales directamente en la GPU. Al calcular la combinación lineal $C=2A+5BC = 2A + 5B$ sobre vectores generados en memoria GPU, se obtuvo un resultado que, al ser comparado con el cálculo equivalente realizado en CPU con NumPy, mostró una diferencia numérica insignificante, validando la equivalencia de los resultados dentro del margen de error aceptado en cálculos en punto flotante. Esto confirma que ElementwiseKernel es una herramienta adecuada para acelerar operaciones elementales sobre grandes volúmenes de datos, reduciendo el tiempo de cómputo y aprovechando el paralelismo masivo que ofrece la arquitectura CUDA.

Cuestionario

1. **¿Cuál es la principal ventaja de utilizar ElementwiseKernel en lugar de escribir un kernel CUDA manual?**
   a) Permite operaciones condicionales complejas
   b) Permite escribir kernels más largos
   c) Facilita operaciones simples sin escribir __global__
   d) Aumenta el uso de CPU

2. **¿Qué papel cumple la variable i dentro de la definición de ElementwiseKernel?**
   a) Es el número total de hilos
   b) Es el índice del bloque CUDA
   c) Representa el índice del hilo que accede a los datos
   d) Es el puntero al kernel

3. **¿Qué tipo de operaciones son ideales para implementarse usando ElementwiseKernel?**
   a) Reducciones paralelas
   b) Multiplicación de matrices
   c) Operaciones por cada elemento independiente del vector
   d) Filtros convolucionales

4. **¿Cuál es el objetivo de usar gpuarray.empty_like(...) en el código?**
   a) Reservar memoria en CPU
   b) Crear una copia de seguridad
   c) Crear un arreglo de salida en GPU con la misma forma y tipo que otro
   d) Convertir un GPUArray a NumPy

5. **¿Qué módulo se utiliza para generar datos aleatorios directamente en la GPU?**
   a) numpy.random
   b) pycuda.gpuarray
   c) pycuda.curandom
   d) pycuda.tools

6. **¿Qué instrucción se usa para verificar que dos resultados GPU y CPU son numéricamente equivalentes?**
   a) assert np.sum(...)
   b) assert la.norm(...).get() < 1e-5
   c) assert x == y
   d) gpuarray.equal(...)

7. **Después de ejecutar una operación GPU, ¿cómo se transfiere el resultado a CPU para impresión o análisis?**
   a) array.numpy()
   b) array.cpu()
   c) array.get()
   d) array.read()

8. **¿Cuál de las siguientes afirmaciones sobre gpuarray es verdadera?**
   a) Es una clase que solo funciona con enteros
   b) Permite realizar operaciones vectorizadas directamente en GPU
   c) Solo sirve para convertir datos de GPU a CPU
   d) No soporta operaciones aritméticas

9. **¿Qué sucede si el tamaño del vector no es múltiplo del número de hilos por bloque en CUDA?**
   a) El kernel falla automáticamente
   b) Se redondea el tamaño y se lanzan hilos innecesarios que deben ser controlados con if (i < n)
   c) Solo se ejecutan los hilos completos
   d) No se puede ejecutar ningún kernel

10. **¿Qué función de PyCUDA permite crear kernels que operan sobre cada elemento sin declarar __global__ ni compilar manualmente?**
    a) SourceModule
    b) RawKernel
    c) ElementwiseKernel
    d) CompileFunction

Referencias

- NVIDIA Corporation. (2023). CUDA C++ Programming Guide . Disponible en: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- generation . Parallel Computing, 38(3), 157–174.
- Kirk, D. B., & Hwu, W. W. (2012). Programming Massively Parallel Processors: A Hands-on Approach . Morgan Kaufmann.
- Sanders, J., & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.
- NumPy Developers. (2024). *NumPy Manual*. https://numpy.org/doc/