1 Función de Correlación en 1D

Dada una señal $x(n)$ y un filtro $h(n)$, la correlación entre ambas $y = x * h$ se puede definir como:

$$y(n) = [x * h](n) = \sum_k x(k+n)h(k),$$

Considerar un array x de números reales de tamaño N que representa la señal de entrada en un dominio discreto, y un array h de tamaño $N_h \ll N$ que describe el filtro h, tal que $h[n] = 0$ *fuera* del intervalo $0 \leq n < N_h$. Para calcular el array $y[n]$ en el intervalo $0 \leq n < N$ se completa ("padding") el array de entrada x con un cierto número de elementos extra, tal que $y[n]$ se pueda calcular desde $n = 0$ hasta $n = N - 1$.

Implementar un kernel conv_gpu que calcule el array y de tamaño N a partir de x y h.

(a) Resuelva los TODO y compruebe que el programa compile (usando el Makefile provisto), corra, y pase exitosamente el test de correctitud.

(b) Usando las herramientas discutidas en clase analize y discuta la performance del código vs tamaño del input y del soporte del filtro. Compare con la performance de la versión en CPU provista (¿es esta última óptima?).

2 Integración numérica

El método de trapecios permite calcular la integral de una función $f(x)$ en un intervalo $[a, b]$ numéricamente. Si se utiliza una grilla uniforme de tamaño $h = (b - a)/n$, la integral es la suma de las áreas de los trapecios $I \approx h \sum_{i=0}^{n-1} I_i$ con $I_j = [f(a + jh) + f(a + (j + 1)h)]/2$.

(a) Construir el kernel (trapeciosGPU) que calcule el array I_j en GPU.

(b) Haga la reducción final de dos formas: (i) Copie I_j al host, y haga la reducción final en CPU para obtener I; (ii) Construya otro kernel (reduGPU) para realizar la reducción de I_j en GPU usando los ejemplos de reducción en árbol vistas en clase. Compare los tiempos de ejecución en casa caso.

(c) Compare el tiempo de ejecución de la versión que hace todo en GPU con el de un programa totalmente serial, versus n. Use los timers provistos en clase, experimente con `nvprof` y `gprof`.

(d) ¿Como generalizaría el código de GPU para una grilla irregular? ($h \to h_j$).

(e) ¿Como haría para fusionar trapeciosGPU y reduGPU en un sólo kernel? Si se le ocurre, impleméntelo y compare el tiempo de integración con el programa de los dos kernels separados.

(f) Piense una implementación paralela del método de Simpson:

$$\int_a^b f(x)dx = \sum_{j=1,3,5,\ldots}^{n-1} \frac{h}{3} [f(x_{j-1}) + 4f(x_j) + f(x_{j+1})] \tag{2.1}$$

3 Reducción y Prefix-Scan: suma de series

Usando la librería Thrust calcular en gpu las series

$$S_1(N) = \sqrt{6 \sum_{i=1}^{N} \frac{1}{i^2}}, \quad S_2(N) = -4 \sum_{i=1}^{N} \frac{(-1)^i}{2i - 1} \tag{3.1}$$

para $N \gg 1$. ¿Convergen? Si es así, ¿a qué valor?

(a) En una primera versión de los códigos generar la secuencia $i = 1, ..., N$, transformarla en los sumandos, y hacer la reducción o el prefix-sum, todo en device.

(b) Optimizar luego el código usando `thrust::make_counting_iterator` y `thrust::transform_reduce`, evitando completamente el uso de `device_vector` intermedios, y fusionando la transformación con la reducción. Comparar las performances de los dos códigos.

(c) Calcular el vector de sumas parciales $S_1(n)$, $S_2(n)$, para $n = 1, ..., N$ usando `thrust::inclusive_scan` y `thrust::make_counting_iterator`.

$\boxed{4}$ **Una transformación iterada y caótica: mapeo logístico**

El mapeo logístico es una relación de recurrencia polinómica de grado 2, que demuestra como un comportamiento caótico complejo puede salir de una dinámica no-lineal extremadamente simple. Fue propuesto en por el biólogo Robert May en 1976 como modelo demográfico en tiempo discreto. Matemáticamente la relación de recurrencia es:

$$x_{n+1} = r x_n (1 - x_n).$$

donde x_n es un numero real y se toma en el intervalo $[0, 1]$ y r, también real, en el intervalo $(0, 4]$. Dependiendo del valor de r, este sistema dinámico discreto puede converger a un punto fijo, a un comportamiento periódico o a uno caótico en función del tiempo discreto n. Este comportamiento puede caracterizarse a través del exponente de liapunov asociada a una trajectoria de $\tau \gg 1$ pasos,

$$\lambda = \sum_{n=0}^{\tau-1} \log(|r - 2 r x_n|).$$

Para ver esto, para múltiples valores distintos de $r[i]$, haremos evolucionar un array de N variables independientes $x[i]$ por un determinado tiempo τ, y calcularemos de paso el exponente de liapunov $\lambda[i]$ asociado a cada trajectoria, para $i = 0, ..., N - 1$.

(a) En el template de programa provisto encontrará una implementación serial completa usando vectores de host. Analícela y escriba una implementación paralela usando vectores de device y algoritmos de `Thrust`.

(b) Mida la aceleración del código para GPU respecto del de CPU.

(c) Para visualizar la solución rapidamente puede usar el comando de gnuplot descripto en el template. Compare con los gráficos resultante con los de:
`https://en.wikipedia.org/wiki/Logistic_map`.

(d) Traduzca todo el programa a CUDA C/C++ escribiendo todos los kernels necesarios.

$\boxed{5}$ **Otra transformación iterada y caótica: Ecuaciones de Lorenz**

Las ecuaciones de Lorenz describen un modelo supersimplificado de los rollos de convección en la atmósfera a través de tres variables dinámicas acopladas en forma no lineal,

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= rx - y - xz \\ \dot{z} &= xy - bz, \end{aligned}$$

donde σ, r, y b son parámetros, y $\dot{u} \equiv du/dt$. Lorenz descubrió (1963) que este sistema determinista y simple puede presentar soluciones que oscilan irregularmente, denominadas caóticas, donde las variables se confinan en una región acotada del espacio con dimensión fractal, llamado atractor caótico.

(a) Resuelva las ecuaciones usando el método de Euler, el cual dada una ecuación $\dot{\vec{r}} = \vec{f}(\vec{r})$, la resuelve iterativamente mediante $\vec{r}_{t+\Delta t} = \vec{r}_t + \Delta t \vec{f}(\vec{r}_t)$, con un Δt suficientemente pequeño. Usando `Thrust` escriba un programa que resuelva esta ecuación para $N \gg 1$ condiciones iniciales simultáneamente, definiendo tres vectores $\{x_t^i, y_t^i, z_t^i\}_{i=0}^{N-1}$. Use paralelización en GPU y en CPU multicore mediante el openmp backend.

(b) Analice las soluciones numéricamente para distintos conjuntos de N condiciones iniciales: (i) Muestre que para $0 < r < 1$ el origen atrae todas las condiciones iniciales (es decir es un punto fijo globalmente estable). (ii) Muestre que para $1 < r < r_H = \sigma(\sigma+b+3)/(\sigma-b-1)$ hay dos puntos fijos estables a los cuales quedan atraídas todas las trayectorias. (iii) Visualize el atractor caótico para $r > r_H$. Analice en particular el caso que estudió Lorenz: $\sigma = 10$, $b = 8/3$, $r = 28$.

Ayudas: Implemente el método iterativo con un loop temporal, y cada paso de Euler implementelo mediante un algoritmo `for_each` o `transform`, sobre una tupla de tres iteradores agrupados con un `zip_iterator` describiendo todas las trayectorias Para más información mire:

- `https://es.wikipedia.org/wiki/Atractor_de_Lorenz`.
- `Strogatz, Non-linear dynamics and Chaos, Capítulo 9`.