Lenguaje de programación para cálculo paralelo.

Andrés Baamonde Lozano (andres.baamonde@udc.es)
Rodrigo Arias Mallo (rodrigo.arias@udc.es)

12 de diciembre de 2014

Índice

1. Introducción

1.1. Propósito y contextualización

En este documento se propone un lenguaje de programación orientado al cálculo numérico. La sintaxis es idéntica a C, con nuevas funcionalidades.

El objetivo consiste en explotar la capacidad de las máquinas modernas de realizar computaciones en paralelo. Tanto de las tarjetas gráficas (GPU) como de los procesadores multinúcleo (CPU). Y emplear esta capacidad para acelerar los procesos de cómputo que sean paralelizables y así hacer una ejecución más eficiente.

En concreto, se introducirán operaciones como el cálculo de productos matriciales, o operaciones vectoriales, realizadas en paralelo implícitamente.

1.2. Lenguajes relacionados

El lenguaje se basará principalmente en C, ya que es un lenguaje que permite la gestión de la memoria manual lo que provoca (si el programador lo hace adecuadamente) una gestión eficiente de la memoria.

C se caracteriza por permitir la codificación de un programa convirtiendo cada línea en unas pocas instrucciones de código máquina, lo cual permite realizar la programación final de un algoritmo, prácticamente describiendo las instrucciones. De esta forma, es el programador quien tiene la responsabilidad de hacer la programación cuidadosa y adecuada.

En este lenguaje se pretende añadir al cálculo eficiente, pero secuencial, que permite el lenguaje C, la posibilidad de realizar uno paralelo, basándose en OpenCL. De esta forma se enfoca el lenguaje al cálculo matemático eficiente.

Algunas ideas provienen también de Matlab, concretamente los operadores propios de imágenes, que se realizarán empleando código con OpenCL. Estos operadores serán transformaciones, umbralizado, suavizado, convoluciones, etc. Que se implementarán directamente en el lenguaje, a modo de funciones. El propósito de estos operadores es el de facilitar el análisis de datos, dentro del mismo lenguaje, para permitir aprovechar el paralelismo.

2. Paradigma

Programación imperativa procedural utilizando funciones y estructurado.

Se descarga un paradigma de más alto nivel como la programación orientada a objetos ya que esa encapsulación no es necesaria. La finalidad de este lenguaje es el cálculo eficiente y en problemas de carácter matemático no es adecuado ese nivel de abstracción.

Los problemas de una programación orientada a objetos, es que los núcleos en los que se van a ejecutar los códigos, han de ser los más sencillos posibles. Pues resultaría en una gran penalización de memoria y tiempo de transferencia operar directamente con objetos en la GPU.

Sería mas complejo para el programador hacerlo a mano, sin un recolector de basura (por ejemplo) que eliminase la memoria que se reservó y ya no está en uso. Sin embargo se considera una eleción de diseño, en virtud de la velocidad de ejecución.

3. Gestión de memoria

La memoria se gestiona de forma manual. Esto quiere decir que para emplear una variable, ha de conocerse su tamaño previamente. Si se trata de vectores o matrices, han de especificarse sus dimensiones. El propósito es, conociendo las dimensiones de una matriz, poder realizar las operaciones en paralelo distribuyendo el trabajo según su tamaño.

El uso y la limpieza de la memoria corren a cargo del programador. Como en C la gestión de la memoria en tiempo de compilación se hará en la inicialización y la asignación dinámica (en tiempo de ejecución) se hará donde el programador considere oportuno.

A la hora de liberar memoria se hará como en C, prescindiremos de cualquier mecanismo interno de gestión de memoria. Sin embargo están a disposición las funciones de la libc, como free, malloc, realloc, o calloc, que permitirán una gestión semi-automática.

La gestión dinámica y automática de la memoria se ha rechazado. Con estos métodos la memoria permanece retenida durante más tiempo que el necesario. Además consumen recursos adicionales y producen pausas, que lo hacen incompatibles con sistemas en tiempo real.

4. Sistema de tipos

Se emplea un tipado estático, deben especificarse todos los tipos de las variables.

En las matrices se deben especificar sus dimensiones y su tipo. Además han de ser homogéneas, esto es, que todos los elementos son del mismo tipo.

Todas las comprobaciones de tipos, se ralizarán en tiempo de compilación por lo que las hará el compilador. Este sistema de tipos incrementa la fiabilidad de los programas procesados. Además, una comprobación exhaustiva, previene los errores.

Los lenguajes de tipado estático incrementan el tiempo de desarrollo. Pero una ventaja de estos lenguajes es que una vez compilados, no se requiere ninguna comprobación en tiempo de ejecución. Por lo que al traducirse directamente a lenguaje máquina y manejar grandes cantidades de datos, permite una mejora de eficiencia frente a los de tipado dinámico.

5. Tipos de dato

5.1. Básicos

- int
- double
- float
- char
- void
- Punteros a tipos básicos (como en C).

5.2. Complejos

Además de los tipos clásicos de C, se añaden matrices y vectores, y un nuevo tipo de números, para los complejos.

- vector
- matrix
- complex

5.3. Modificadores de tipos

Dada la arquitectura actual de los ordenadores, existen varias memorias en los que se permiten almacenar variables. Algunas permiten compartir la misma memoria RAM, con la GPU, haciendo innecesaria la transmisión de datos a la tarjeta.

Para indicar que una variable se puede usar en esta memoria compartida, se emplea el modificador «shared». Este indica al compilador que, en caso de resultar eficiente, no copie la variable a la memoria interna de la GPU, si no que la lea y modifique directamente en la RAM.

Por defecto las variables son enviadas a las memorias de los distintos procesadores, donde se realizan las operaciones, y posteriormente enviadas de vuelta a la memoria principal.

6. Operadores

6.1. Tabla de operadores

Para utilizar la GPU al máximo, lo más adecuado sería utilizar una tabla de operadores como la de OpenCL. A mayores se incorporarán las funciones típicas de imágenes, como convoluciones y filtros. En estos últimos se podrán aprovechar al máximo las propiedades de las gráficas como el acceso a la memoria de los núcleos vecinos.

add	+
substract	-
multiply	*
divide	/
remainder	%
unary plus	+
unary minus	-
post and pre increment	++
post and pre decrement	–
relational greater than	>
relational less than	<
relational greater-than or equal-to	>=
relational less-than or equal-to	<=
equal	==
not equal	!=
bitwise and	&
bitwise or	|
bitwise not	^
bitwise not	~
logical and	&&
logical or	||
logical exclusive or	^^
logical unary not	!
ternary selection	?:
right shift	»
left shift	«
size of	sizeof
comma	,
dereference	*
address-of	&
assignment	=

6.2. Sobrecarga operadores

6.2.1. Introducción

Se añaden (sobrecargando los operadores) las operaciones con vectores y matrices. El compilador podrá detectar posibles errores gracias al tipado estático.

6.2.2. Operadores vectoriales

Suma, multiplicación y resta. Estos operadores funcionarían con los tipos vector y matriz, pero también para su multiplicación por un escalar.

7. Visibilidad

7.1. Visibilidad local

A nivel de función, la variable es accesible y puede ser manipulada sólo en la función que es declarada.

7.2. Visibilidad global

A nivel de programa, la variable es accesible y puede ser manipulada en todo el programa.

8. Tratamiento de errores

Al ejecutar código en paralelo, ha de tenerse especial cuidado al realizar paradas de emergencia. En una CPU basta con interrumpir el proceso actual. Sin embargo para detener la ejecución de muchos hilos en paralelo, ha de coordinarse una parada global.

Para ello, el compilador añade un fragmento de código en el programa, que se ejecuta cuando este recibe una interrupción, o una señal de parada. Este código permite detener todas las ejecuciones pendientes, y terminar el programa.

8.1. Matemáticos

- Operaciones matemáticas que provocan una pérdida de precisión superior a un umbral prefijado.
- Raíz de un número negativo (sin ser el destino un número imaginario).

8.2. Desbordamiento

Estos errores se producen cuando un programa no controla adecuadamente la cantidad de datos que se copian sobre un área de memoria reservada a tal efecto. Si esa cantidad es superior a la anteriormente asignada los bytes se almacenan contiguamente pisando el contenido que pudiera existir previamente.

Para evitar estos errores, el compilador realiza una tarea de comprobación. Consiste en los siguientes pasos. Primero coloca una constante en el espacio contiguo, posterior al área de destino. Después procede a ejecutar el código de escritura. Luego comprueba que el valor de la constante resulte ser el que había antes de la escritura, y si es así, se deduce que no se ha sobreescrito ningún valor fuera del área especificada. En caso contrario lanza una excepción que aborta el programa.

Ejemplo sin protección:

Antes de la escritura:

```
[ variable A ][ variable B][ variable C ]
[000000000000][11111111111][222222222222]
```

Después, con desbordamiento. La variable B ha sido modificada:

```
[ variable A ][ variable B][ variable C ]
[777777777777][77777777111][222222222222]
```

Ejemplo con protección:

Antes de la escritura:

```
[ variable A ][ A? ][ variable B][ B? ][ variable C ][ C? ]
[000000000000][1234][11111111111][1234][222222222222][1234]
```

Después, con desbordamiento:

```
[ variable A ][ A? ][ variable B][ B? ][ variable C ][ C? ]
[777777777777][7777][77771111111][1234][222222222222][1234]
```

Después de la escritura se comprueba que el valor «A?» coincida con el anterior, en este caso era «1234». Como no coincide, se ha detectado un desbordamiento, y se aborta el programa.

8.3. Segmentación

Se lanza una excepción que interrumpe el programa automáticamente (Lo hace el compilador). Errores :

- Desreferenciación de punteros NULL.
- Intento de acceder a memoria que el programa no tiene permisos.
- Intentar acceder a una dirección de memoria inexistente.
- Intentar escribir memoria de sólo lectura.
- Un desbordamiento de búfer.
- Uso de punteros no inicializados.

9. Características

9.1. Inserción de código OpenCL

Además de la programación normal del lenguaje, se permitirá añadir trozos de código directamente en OpenCL, que nos permiten manejar la GPU manualmente y así optimizar más nuestro código.

También se incluyen operadores para la sincronización entre hilos y bloqueos para la posible concurrencia a una misma variable. Tarea que el planificador en las operaciones implementadas hará automáticamente.

9.2. Evaluación en cortocircuíto

La evaluación de cortocircuito, es algo que el lenguaje de programación incorpora ya que es de gran utilidad ya que evita calculos innecesarios.

Esta evaluación es sumamente sencilla de implementar:

Para el operador AND se evalúa el primer operando y se pospone la evaluación de los siguientes ya que si este primero es falso, da igual el resultado de los siguientes ya que, el resultado total de la sentencia será falso. Y así sucesivamente.

Para el operador OR se evaluará el primer operando y se pospone la evaluación de los siguientes, ya que, si este es verdadero el resultado total de la sentencia será verdadero y no será necesario seguir evaluándola.

Se podría implementar (por versatilidad) con otros operadores la «eager evaluation», que sería evaluar la sentencia entera siempre. Su implementación consistiría en añadir un punto detrás del operador en cuestión, por ejemplo:

```
if((a |. b) &. c)
```

10. Programas o trozos de código

A continuación se muestran algunos programas sencillos, para la operación de vectores, matrices e imágenes. Primero se muestra el código en C original, luego la versión en nuestro lenguaje, y por último, el código OpenCL que implementa la parte paralela.

10.1. Suma de vectores

Ejemplo sin paralelizar:

```
#define N 100

float A[N], B[N], C[N];
int i;

for(i = 0; i < N; i++)
    C[i] = A[i] + B[i];
```

Ejemplo paralelizado:

```
#define N 100

vector float A[N], B[N], C[N];

C = A + B;
```

Ejemplo de kernel para la suma de vectores en OpenCL:

```
__kernel void vector_add(__global const int *A, __global const int *B,
   __global int *C)
{
    int i = get_global_id(0);

    C[i] = A[i] + B[i];
}
```

10.2. Multiplicación de matrices

Ejemplo sin paralelizar:

```
#define N 100

float A[N][N], B[N][N], C[N][N];
float sum = 0.0;
int i, j, k;

for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        sum = 0;
        for(k = 0; k < widthB; k++)
            sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }

}
```

Ejemplo paralelizado:

```
#define N 100

matrix float A[N][N], B[N][N], C[N][N];

C = A * B;
```

Ejemplo de kernel para la multiplicación en OpenCL:

```
__kernel void matrix_mul(__global float* A, __global float* B,
    __global float* C,  int widthA, int widthB )
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    float value=0;
    for(int k = 0; k < widthA; k++)
    {
        value = value + A[k + j * widthA] * B[k*widthB + i];
    }
    C[i + widthA * j] = value;
}
```

10.3. Operador Sobel en imágenes

Porción de código secuencial del algoritmo de Sobel:

```
for (i = mr; i < 256 - mr; i++)
{
    for(j = mr; j < 256 - mr; j++)
    {
        sum1 = 0;
        sum2 = 0;
        for(p = -mr; p <= mr; p++)
        {
            for(q = -mr; q <= mr; q++)
            {
                sum1 += pic[i+p][j+q] * maskx[p+mr][q+mr];
                sum2 += pic[i+p][j+q] * masky[p+mr][q+mr];
            }
        }
        outpicx[i][j] = sum1;
        outpicy[i][j] = sum2;
    }
}
```

Código de ejemplo paralelizado, usando los operadores de imágenes:

```
matrix uint8 I[1024][768], S[1024][768];

S = sobel(I, 100);
```

Código OpenCL para la ejecución paralela del filtro Sobel.

```
__kernel
__attribute__((task))
```

```
void sobel(global unsigned int * restrict frame_in, global unsigned int *
   restrict frame_out,
           const int iterations, const unsigned int threshold)
{
    // Filter coefficients
    int Gx[3][3] = {{-1,-2,-1},{0,0,0},{1,2,1}};
    int Gy[3][3] = {{-1,0,1},{-2,0,2},{-1,0,1}};

    // Pixel buffer of 2 rows and 3 extra pixels
    int rows[2 * COLS + 3];

    int count = 0;
    while (count != iterations) {
        // Each cycle, shift a new pixel into the buffer.
        // Unrolling this loop allows the compile to infer a shift
           register.
        #pragma unroll
        for (int i = COLS * 2 + 2; i > 0; --i) {
            rows[i] = rows[i - 1];
        }
        rows[0] = frame_in[count];

        int x_dir = 0;
        int y_dir = 0;

        // With these loops unrolled, one convolution can be computed
           every
        // cycle.
        #pragma unroll
        for (int i = 0; i < 3; ++i) {
            #pragma unroll
            for (int j = 0; j < 3; ++j) {
                unsigned int pixel = rows[i * COLS + j];
                unsigned int b = pixel & 0xff;
                unsigned int g = (pixel >> 8) & 0xff;
                unsigned int r = (pixel >> 16) & 0xff;

                // RGB -> Luma conversion approximation
                // Avoiding floating point math operators greatly reduces
                // resource usage.
                unsigned int luma = r * 66 + g * 129 + b * 25;
                luma = (luma + 128) >> 8;
                luma += 16;

                x_dir += luma * Gx[i][j];
                y_dir += luma * Gy[i][j];
            }
        }

        int temp = abs(x_dir) + abs(y_dir);
        unsigned int clamped;
        if (temp > threshold) {
            clamped = 0xffffff;
        } else {
            clamped = 0;
```

```
        }

        frame_out[count++] = clamped;
    }
}
```

11. Pruebas de ejecución

A continuación se muestran algunos resultados reales, para la multiplicación de matrices, al ejecutarse en secuencia, o en paralelo. La operación que se realiza es la multiplicación de matrices antes mencionada.

```
Tamano de matriz: 64x64
GPU envio:      4.070000E+02 us
GPU comp.:      1.159000E+03 us
GPU recep.:     3.410000E+02 us
GPU total:      2.148000E+03 us
GPU total:      2148.000000 us

CPU comp.:      3.190000E+02 us
CPU comp.:      319.000000 us

Aceleracion:    0.148510
Fallos comp.:   0
Max err.:       0.000000E+00
----------------------------------------
Tamano de matriz: 512x512
GPU envio:      2.384000E+03 us
GPU comp.:      3.621100E+04 us
GPU recep.:     1.562000E+03 us
GPU total:      4.046600E+04 us
GPU total:      40466.000000 us

CPU comp.:      3.352140E+05 us
CPU comp.:      335214.000000 us

Aceleracion:    8.283843
Fallos comp.:   0
Max err.:       0.000000E+00
----------------------------------------
Tamano de matriz: 1024x1024
GPU envio:      5.669000E+03 us
GPU comp.:      3.109710E+05 us
GPU recep.:     5.350000E+03 us
GPU total:      3.222790E+05 us
GPU total:      322279.000000 us

CPU comp.:      1.205548E+07 us
CPU comp.:      12055485.000000 us

Aceleracion:    37.406983
Fallos comp.:   0
Max err.:       0.000000E+00
----------------------------------------
```

```
Tamano de matriz: 2048x2048
GPU envio:      2.168600E+04 us
GPU comp.:      2.418712E+06 us
GPU recep.:     1.743300E+04 us
GPU total:      2.458115E+06 us
GPU total:      2458115.000000 us = 2.46 s

CPU comp.:      1.115366E+08 us
CPU comp.:      111536649.000000 us = 111.5 s = 1 min 51 s

Aceleracion:    45.374870
Fallos comp.:   0
Max err.:       0.000000E+00
```

Cuanto más grande sea la matriz o el vector, más se aprecia la aceleración en paralelo. También se observa que el tiempo de transferencia aumenta considerablemente con el tamaño de la matriz.

Para una comprobación posterior, en cada simulación se comprueba que ambas matrices (la calculada con la CPU y GPU), coincidan.

12. Necesidades especiales

12.1. IDE

Un IDE con todo lo que el lenguaje necesitase para ser elaborado (editor de código, un compilador y un depurador) sería lo ideal para el desarrollo del lenguaje ya que agilizaría el codificado de nuestros programas.

Gráficamente, sería del estilo de los IDEs clásicos pero con una gran similitud al de Matlab ya que, para las matrices el Workspace de variables de matlab, junto con la consola interactiva, el debug de cosas sencillas es extremadamente útil.

También cabría la posibilidad de que, en lugar de tener un IDE propio, construir un plugin para un IDE conocido.

Además, para el diseño de algoritmos paralelos, sería adecuado disponer de alguna herramienta que permita dibujar a modo de grafo, la ejecución en paralelo o secuencial de las distintas partes del código. Aunque queda fuera del alcance de este documento.

12.2. Debug

Se puede compilar de forma secuencial en la CPU para así poder depurar sin problemas en algoritmo en una primera versión y después poder optimizar el cálculo haciéndolo de forma paralela. Esto se propone debido a que el debug de un procesamiento paralelo es tedioso.

12.3. Puntos de interrupción

Al parar el proceso podemos inspeccionar el valor de las variables, los argumentos de la pila, y las direcciones de memoria sin que el proceso modifique estos valores hasta que no se lo indicase de esa forma el depurador.

También podemos inspeccionar el valor de los registros del procesador e incluso podemos cambiar cualquier dato tanto en los registros como en la memoria si queremos.

Los breakpoints son sin duda la característica más comúnmente utilizada en un depurador. Por lo que serán incorporados al depurador de nuestro lenguaje. Existen dos tipos de breakpoints:

- software breakpoints (se usan específicamente para detener la CPU cuando se ejecuta una instrucción)
- hardware breakpoints (a nivel de procesador no implementados por un lenguaje)

12.4. Dispositivos para el procesado paralelo

Está claro que para realizar cálculos en paralelo necesitaremos contar con más de un núcleo, para procesar varias instrucciones a la vez.

Para ello los ordenadores actuales cuentan con un procesador, que suele disponer de varios núcleos, y también una o más tarjetas gráficas o GPU.

Además todos ellos deben ser compatibles con OpenCL. Para ello el fabricante ha de proveer la implementación que corresponde para dicho dispositivo. Sin embargo, la mayoría de fabricantes incluyen OpenCL en todos sus dispositivos actuales.

13. Arquitectura

13.1. Compilador

Ya que el lenguaje es una mejora de C, sería recomendable basarse en un compilador de C existente, al que añadir las mejoras.

Para compilar el código paralelo, es necesario primero identificar aquellos fragmentos que sean paralelizables. También es importante detectar que zonas han de terminar su ejecución antes que otras, por lo que debe construirse un grafo de ejecución.

En este grafo, los nodos forman las instrucciones o grupos de instrucciones que se ejecutan como un bloque. Y las aristas forman un árbol de dependencias, que representan la necesidad de cubrir los nodos anteriores antes de ejecutar el actual.

Estos grafos permiten la optimización de código e instrucciones inútiles. Por ejemplo, incrementar una variable en un bucle que nunca se usa.

Los bloques que se ejecutarán en paralelo, son tratados de forma especial. Se divide el bloque en partes. Se crea una cola de trabajo donde se introducen todas las partes que deben ser procesadas. A cada núcleo se le asigna una parte. De forma que cuando alguno termine, se le asigna la siguiente parte que contiene la cola. El proceso termina cuando la cola está vacía, y todos los núcleos han terminado su ejecución.

El codigo paralelo es especificado por el programador a través de los tipos especiales para matrices y vectores.

Para la realización de operaciones entre tipos diferentes, el compilador realiza una conversión automática, siempre que sea posible. Por ejemplo de entero a flotante.

13.2. Errores

Cuando se produce una excepción en la CPU, es fácil detener la ejecución del programa, si éste es secuencial. Sin embargo, cuando se trabaja con varios hilos de procesamiento, la excepción en uno de ellos, implica la detención de todos los demás.

Esto se realiza empleando un sistema de bloqueos, que detiene todo el programa, y espera a que todos los núcleos hayan terminado y se hayan detenido antes de la terminación del programa principal.

A pesar de que con los lenguajes orientados a objetos las excepciones están substituyendo a los códigos de error, nosotros no implementaremos estas excepciones.

Cuando una excepción se produce, es el hardware el que avisa de tal evento, y empleando una tabla de excepciones, el CPU o GPU, es la encargada de transmitir el tipo de excepción, que nuestro lenguaje recogerá, y abortará el programa adecuadamente.

14. Fuentes

- Funcionamiento OpenCL
- Operadores OpenCL
- Transformar operaciones de Matrices para GPU
- Funcionamiento GPU
- Modificadores de tipos
- Tipos de datos Escalares en OpenCL
- Tiposde datos para Vectores en OpenCL
- Visibilidad OpenCL
- Eager Evaluation
- Lazy Evaluation(cortocircuíto)