Cuda GPU Dynamically Resizable Array

Enzo Meneses M.[1]

Universidad Austral de Chile, Valdivia, Chile

Abstract. En este trabajo se presentan los primeros avances en la implementación de un arreglo dinámico paralelo para GPU utilizando cuda, teniendo en cuenta solo la operación push_back. Según el conocimiento del autor es la primera vez que esto se intenta. La implementación se basa fuertemente en LFVector, que es el primer intento de un arreglo dinámico paralelo y que ha sido una gran influencia en investigación posterior.
Primero se hizo una implementación directa de LFVector a cuda, que producto de las diferencias hardware no dio buenos resultados. El segundo metodo utiliza la técnica busy-waiting para solucionar los problemas presentados por LFVector. Esto resulta exitosamente, pero con una limitación importante. Este método funciona solo con un único bloque de threads.

Keywords: cuda · gpu · dynamic array.

1 Introducción

Con el paso de los años y el desarrollo de las GPUs, cada vez es mas común el uso de estas en para el computo de tareas que requieren de un gran trabajo, por ejemplo simulación de procesos físicos. Sin embargo, para lograr un mayor rendimiento las GPUs también añaden ciertas restricciones que vienen de la mano con el procesamiento altamente paralelo. Estas restricciones provocan que ciertas tareas sean mucho mas difíciles de implementar que en forma secuencial o incluso paralelismo en CPU. Uno de estos casos es la inserción de elementos en paralelo en un arreglo. EN GPU normalmente se trabaja con estructuras estáticas y bien estructuradas, que presentan las mayores ganancias con la paralelización.

Sin embargo, la mayoría de problemas reales suelen tener comportamientos mas dinámicos y y requieren de un mayor esfuerzo del investigador para implementar algoritmos que incorporen este dinamismo. Además, estas soluciones suelen ser solo aplicables al problema para el que fueron propuestas, por lo que cada problema dinámico requiere nuevamente esfuerzo extra para solucionar algo que no es realmente parte de lo que se busca investigar.

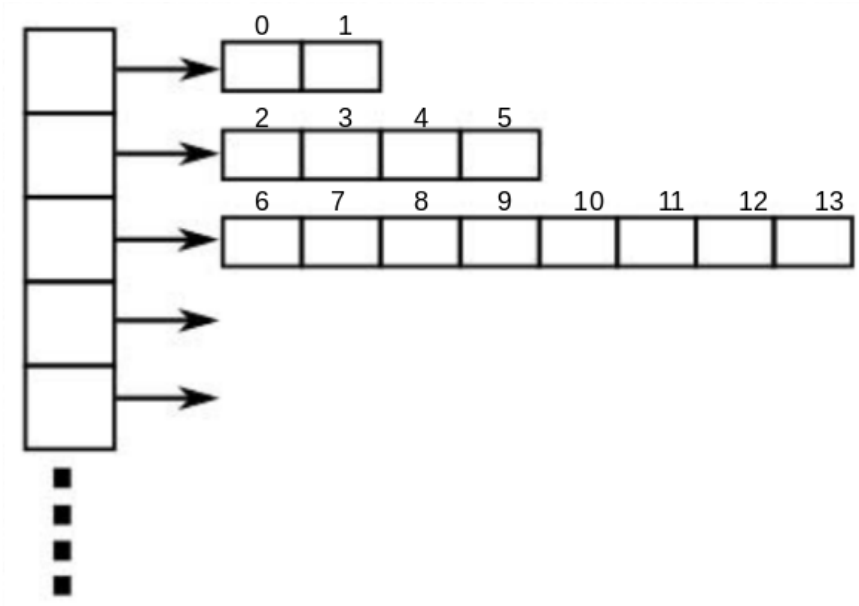
Si bien hay bastante investigación sobre estructuras dinámicas en paralelo, incluyendo arreglos, estas se centran en estructuras y algoritmos para CPUs. Al momento de escribir esto no sido posible encontrar algún intento de implementar un arreglo dinámico en GPU que sirva como solución general para distintos tipos de problemas.

2 Implementación

2.1 LFVector

El primer intento de un arreglo dinámico paralelo fue realizado por Damian Dechev[1] en el 2006, con la intención de crear una estructura similar al vector de la librería estándar de c++, pero que pueda ser utilizado en paralelo. Si bien la investigación de estructuras paralelas comenzó mucho antes, esta investigación es la base que ha influido a la mayoría de investigaciones posteriores sobre arreglos dinámicos en paralelo.

Damian Dechev propone una idea similar a la de los doubling arrays, pare adaptada de tal forma que el incrementar el tamaño, no sea necesario mover los elementos ya existentes a un espacio nuevo de memoria. Esto es muy importante, ya que evita la necesidad de coordinar los distintos threads para que ninguno realice operación sobre el arreglo antiguo una vez movido los elementos. La estructura que propone consiste en un arreglo de punteros, donde cada puntero apunta a un arreglo del doble del tamaño que el del puntero anterior. Y cada vez que se llena se reserva la cantidad de memoria que corresponda al siguiente puntero, duplicando la capacidad del arreglo.



En este trabajo se va a implementar solo la operación push_back y las necesarias para que esta funcione. Esto es posible en cuda de una forma bastante directa, con la excepción de la función Compare_And_Swap (CAS), ya que la función atomicCAS de cuda esta definida solo para algunos tipos de datos, para

lo que se debe usar una técnica llamada type punning o reinterpret_cast en c++ moderno.

Para insertar un elemento al final, primero se realiza la operación Fetch_And_Add (atomicAdd en cuda), para incrementar el tamano del arreglo y obtener la posición en la que se debe insertar de forma atomica. Después se reserva mas memoria si es necesario y finalmente se guarda el valor en la posición correspondiente. La parte mas complicada es la de reservar memoria, ya que hay muchos threads ingresando elementos al mismo tiempo y si alguno intenta ingresarlo antes de que se reserve la memoria, ocurre un error.

Para evitar esto todos los threads reservan la cantidad de memoria necesaria e intentan asignarla con atomicCAS al puntero correspondiente, de modo que el primero que llega asigna la memoria al puntero y el resto puede liberar su memoria reservada y utilizar la que reservo el primero.

```
struct LFVector {
        unsigned int size;
        int **a;
        LFVector();
        __device__ void resize(unsigned int n);
        __device__ int& at(unsigned int i);
        __device__ int get_bucket(unsigned int i);
        __device__ void new_bucket(unsigned int b);
        __device__ void push_back(int e);
};

__device__ void LFVector::new_bucket(unsigned int b) {
        int bsize = 1 << (5 + b);
        int *aux = (int*)malloc(sizeof(int) * bsize);
        int *nil = nullptr;
        int **addr = &(a[b]);
        ull old = atomicCAS((ull **) &addr,
                            (ull *) &nil,
                            (ull *) &aux);
        if ((* (int **) &old) != nullptr) {
                free(aux);
        }
}

__device__ void LFVector::push_back(int e) {
        int idx = atomicAdd(&size, 1);
        int b = get_bucket(idx);
        if (a[b] == nullptr) {
                new_bucket(b);
        }
        at(idx) = e;
}
```

El problema de este metodo, es que en una GPU, que es altamente paralela, hay muchos threads reservanod memoria para intentar asignarla, mas memoria de la que es posible reservar, por lo que en el caso de cuda un trhead puede terminar correctamente la funcion new_bucket, pero sin reservar memoria nueva, lo que eventualmente produce un error.

2.2 Busy-waiting

Para resolver lo anterior se agrego un nuevo arreglo a la estructura que indica si un bloque de memoria ha sido asignado. De esta forma es posible hacer el atomicCAS sobre ese arreglo y que solo un thread se encargue de reservar y asignar memoria. Para evitar que otros threads intentan ingresar un elemento antes de que se termine de reservar la memoria, se agrego un loop del que los threads no pueden salir hasta que este asignado el bloque en el que necesitan insertar un elemento.

Esto normalmente seria un problema en CPU y que normalmente se intenta evitar, ya que debido a las políticas que se usan para ejecutar múltiples threads en un solo núcleo, es posible que el thread encargado de reservar memoria detenga su ejecución indefinidamente provocando que todos los otros threads queden indefinidamente en el loop sin poder progresar.

Sin embargo en GPU, dentro de un warp todos los threads avanzan coordinadamente, por lo que no es posible que ocurra lo anterior. Aunque, este metodo si se queda bloqueado al ejecutarse con mas de un bloque. Pero funciona correctamente con un solo bloque con múltiples warps, algo que en ciertas situaciones es compromiso aceptable.

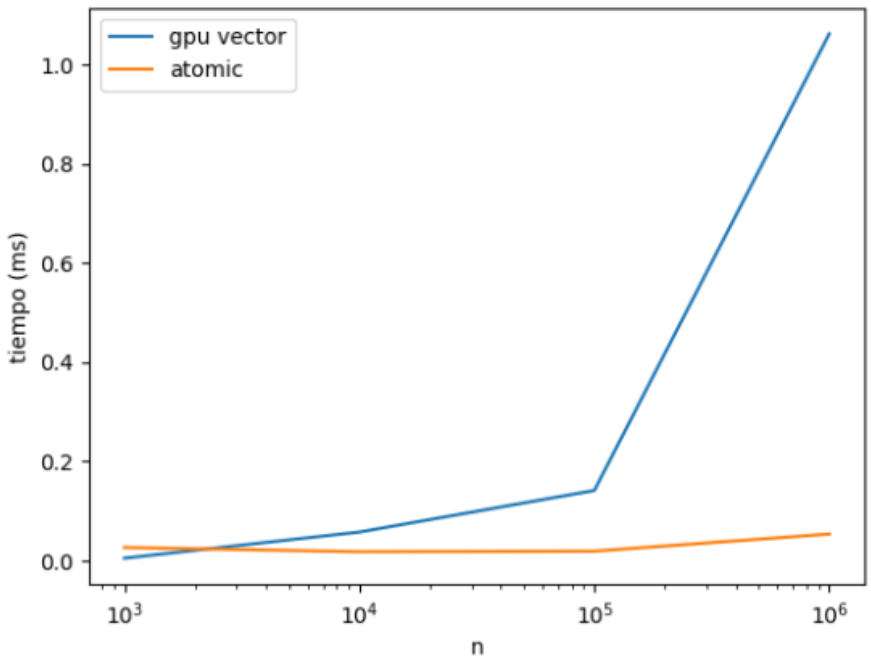
```
struct LFVector {
        unsigned int size;
        int **a;
        int *isbucket;
        LFVector();
        __device__ void resize(unsigned int n);
        __device__ int& at(unsigned int i);
        __device__ int get_bucket(unsigned int i);
        __device__ void new_bucket(unsigned int b);
        __device__ void push_back(int e);
};

__device__ void LFVector::new_bucket(unsigned int b) {
        if (atomicCAS(isbucket + b, 0, 1) == 0) {
                int bsize = 1 << (5 + b);
                a[b] = (int*)malloc(sizeof(int) * bsize);
        }
}

__device__ void LFVector::push_back(int e) {
```

```
        int idx = atomicAdd(&size, 1);
        int b = get_bucket(idx);
        while (a[b] == nullptr) {
            new_bucket(b);
        }
        at(idx) = e;
}
```

3 Resultados

El primer método solo funciona de forma fiable con un N ¡ 1000 en guanaco, que es muy bajo como para hacer pruebas de su rendimiento.

En cuanto al segundo método, la prueba consistió en crear un arreglo con enteros en el rango [0,99] e insertar una copia del elemento en la posición i, si su valor es menor a 90. Para esto, se usaron tamaños entre 10^3 y 10^6 con un bloque de 1024 threads que itera sobre el arreglo realizando la operación push_back cuando corresponde. Su rendimiento se puede apreciar en el siguiente gráfico.



La linea naranja corresponde a la misma prueba, pero con toda la memoria necesaria reservada de antemano y realizando la operación push_back solo con

atomicAdd para obtener el índice donde insertar e insertando el elemento directamente en esa posición, representando un limite inferior para la solución del problema.

4 Conclusión

Al terminar este reporte, todavía queda mucho trabajo por hacer para obtener un arreglo dinámico en GPU de uso general. Sin embargo, con el trabajo realizado quedaron mucho mas claras las limitaciones del problema en maquinas altamente paralelas. También, cabe destacar que a pesar de tener una limitación muy importante, como el uso de un solo bloque de trhreads, el objetivo fue logrado parcialmente.

En cuanto a las siguientes proposiciones de solución, queda bastante claro que una opción viable es dividir el problema en bloques independientes que correspondan a los bloques de threads, permitiendo trabajar localmente con un gran grado de sincronización, y utilizar alguna estructura global con la información de los distintos bloques para que pueda ser utilizado como si fuese un solo arreglo compacto.

References

1. Dechev, D., Pirkelbauer P. and Stroustrup B.: Lock-Free Dynamically Resizable Arrays. OPODIS (2006).