Programación de GPUs en CUDA: Optimización de aplicaciones 3

Pedro Alonso

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Contenido

Lecturas vectorizadas

Introducción

- Muchos kernels CUDA están limitados por el ancho de banda de acceso a memoria.
- Para mitigar los cuellos de botella de ancho de banda en el código se pueden usar cargas y almacenamientos vectoriales en CUDA.

Sea por ejemplo el siguiente código:

Lectura y escritura de vectores

```
__global__ void device_copy_scalar_kernel( int* d_in, int* d_out, int N ) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for ( int i = idx; i < N; i += blockDim.x * gridDim.x ) {
    d_out[i] = d_in[i];
  }
}
void device_copy_scalar( int* d_in, int* d_out, int N ) {
  int threads = 128;
  int blocks = min((N + threads-1) / threads, MAX_BLOCKS);
  device_copy_scalar_kernel<<< blocks, threads >>>( d_in, d_out, N );
}
```

Fuente: https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/

Instrucciones vectorizadas

- En el código anterior existen seis instrucciones asociadas con la operación de copia: Cuatro instrucciones para calcular direcciones de carga y almacenamiento, una lectura y una escritura de 32 bits.
- Se puede mejorar el rendimiento utilizando instrucciones de carga y almacenamiento vectorizadas, es decir, que cargan y almacenan datos con anchos de 64 o 128 bits.
- El uso de cargas vectorizadas reduce el número total de instrucciones, la latencia y mejora la utilización del ancho de banda.
- La forma más sencilla de usar cargas vectorizadas es usar los tipos de datos vectoriales definidos en los ficheros cabecera de CUDA C/C++, como `int2`, `int4` o `float2`.
- Estos tipos se pueden usar fácilmente mediante la conversión de tipos en C/C++. Por ejemplo, en C++ se puede convertir el puntero `int d_in` en un puntero `int2` usando `reinterpret_cast<int2*>(d_in)` (o `(int2*(d_in))`).
- Cambiar el tipo de dato de estos punteros hará que el compilador genere las instrucciones vectorizadas. Sin embargo, hay que tener en cuenta que estas instrucciones requieren datos alineados en memoria.

Instrucciones vectorizadas

Lectura y escritura de vectores, primera versión vectorizada

```
__global__ void device_copy_vector2_kernel( int* d_in, int* d_out, int N ) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for ( int i = idx; i < N/2; i += blockDim.x * gridDim.x ) {
    reinterpret_cast<int2*>(d_out)[i] = reinterpret_cast<int2*>(d_in)[i];
  }

  // in only one thread, process final element (if there is one)
  if ( idx==N/2 && N%2==1 )
    d_out[N-1] = d_in[N-1];
}

void device_copy_vector2( int* d_in, int* d_out, int n ) {
  threads = 128;
  blocks = min((N/2 + threads-1) / threads, MAX_BLOCKS);

  device_copy_vector2_kernel<<< blocks, threads >>>( d_in, d_out, N );
}
```

- El compilador genera las mismas seis instrucciones pero la lectura y escritura son de 64 bits.
- El número de instrucciones es de la mitad: el bucle solo se ejecuta N/2 veces.

Instrucciones vectorizadas

Lectura y escritura de vectores, segunda versión vectorizada

```
__global__ void device_copy_vector4_kernel( int* d_in, int* d_out, int N ) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for( int i = idx; i < N/4; i += blockDim.x * gridDim.x ) {
    reinterpret_cast<int4*>(d_out)[i] = reinterpret_cast<int4*>(d_in)[i];
  }

  // in only one thread, process final elements (if there are any)
  int remainder = N%4;
  if ( idx==N/4 && remainder!=0 ) {
    while( remainder ) {
      int idx = N - remainder--;
      d_out[idx] = d_in[idx];
    }
  }
}

void device_copy_vector4( int* d_in, int* d_out, int N ) {
  int threads = 128;
  int blocks = min((N/4 + threads-1) / threads, MAX_BLOCKS);

  device_copy_vector4_kernel<<< blocks, threads >>>( d_in, d_out, N );
}
```

Atómicos a nivel de Warp

Introducción

- Las operaciones *atómicas agregadas por Warp* constituyen una técnica útil para mejorar el rendimiento cuando muchos threads suman atómicamente a un único contador.
- En la agregación por *Warp*, los threads de un *Warp* primero calculan un incremento total entre sí y luego eligen un solo thread para que sume atómicamente el incremento a un contador global.
- Esta agregación puede llegar a reducir hasta 32 veces el número de operaciones atómicas realizadas por un *Warp*.

Introducción: *stream compaction*

- Ejemplo: Se dispone de un array, `src`, con `n` elementos y un predicado, se necesita copiar todos los elementos de `src` que satisfacen el predicado en el array de destino, `dst`.
- Para simplificar, supongamos que `dst` tiene una longitud de al menos `n` y que el orden de los elementos en el array `dst` no importa.

Problema de filtrado de elementos en un vector

```
int filter(int *dst, const int *src, int n) {
  int nres = 0;
  for (int i = 0; i < n; i++)
    if (src[i] > 0)
      dst[nres++] = src[i];
  // return the number of elements copied
  return nres;
}
```

Introducción: *stream compaction*

- Solución con atómicos en memoria global:

Filtrado de elementos en un vector: atómicos sobre memoria global

```
__global__ void filter_k( int *dst, int *nres, const int *src, int n ) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if( i < n && src[i] > 0 )
    dst[ atomicAdd( nres, 1 ) ] = src[ i ];
}
```

- Problema: El grado de colisiones de `atomicAdd()` es alto, lo que limita el rendimiento.

Introducción: *stream compaction*

Filtrado de elementos en un vector: atomicos sobre memoria compartida

```
__global__ void filter_shared_k( int *dst, int *nres, const int* src, int n ) {
  __shared__ int l_n;
  int i = blockIdx.x * (NPER_THREAD * BS) + threadIdx.x;

  for ( int iter = 0; iter < NPER_THREAD; iter++ ) {
    if (threadIdx.x == 0)
      l_n = 0;
    __syncthreads();

    int d, pos;
    if( i < n ) {
      d = src[i];
      if( d > 0 )
        pos = atomicAdd( &l_n, 1 );
    }
    __syncthreads();

    if( threadIdx.x == 0 )
      l_n = atomicAdd( nres, l_n );
    __syncthreads();

    if( i < n && d > 0 ) {
      pos += l_n; // increment local pos by global counter
      dst[pos] = d;
    }
    __syncthreads();
    i += BS;
  }
}
```

Solución: *Warp-Aggregated Atomics*

- La *agregación de Warp* es el proceso de combinar operaciones atómicas de varios threads en uno solo.
- Con la agregación de *Warp*, reemplazamos las operaciones atómicas con los siguientes pasos:
  1. Los threads en el *Warp* eligen un thread líder.
  2. Los threads en el *Warp* calculan el incremento atómico total del warp.
  3. El thread líder realiza una suma atómica para calcular el desplazamiento del warp.
  4. El thread líder transmite el desplazamiento a todos los demás threads en el warp.
  5. Cada thread agrega su propio índice dentro del *Warp* al desplazamiento del warp para obtener su posición en la matriz de salida.
- A partir de CUDA 9.0, existen dos API: Grupos Cooperativos y funciones primitivas síncronas con *Warp*.
- Tras ejecutar una operación atómica agregada con *Warp*, cada thread procede como en el código original y escribe su valor en su posición en la matriz `dst`.

Solución: *Warp-Aggregated Atomics*

1. Elección del líder.

- Generalmente, se debe asumir que solo algunos threads están activos, por lo que se necesita un grupo formado por todos los threads activos.
- Para utilizar grupos cooperativos se debe incluir el archivo cabecera y utilizar el espacio de nombres adecuado.

```
#include <cooperative_groups.h>
using namespace cooperative_groups;
```

- A continuación se crea un grupo formado por los threads coalescentes actuales (los que están activos en ese momento):

```
auto g = coalesced_threads();
```

- Cada thread obtiene su "rango":

```
int th_rk = g.thread_rank();
```

Solución: *Warp-Aggregated Atomics*

2. Cálculo del incremento total del *Warp*.

- En el ejemplo de *filtrado*, cada thread con un predicado verdadero incrementa el contador en 1.
- El incremento total de la trama es igual al número de vías (*lane*) activas (no se considera aquí el caso de incrementos que varían entre carriles).
- Esto es trivial con los Grupos Cooperativos:

```
g.size()
```

devuelve el número de threads en el grupo.

3. Realizar la suma atómica.

- Solo el thread líder (vía 0) realiza la operación atómica.
- Con grupos cooperativos es simplemente verificar si `thread_rank()` devuelve 0.

```
int warp_res;
if( g.thread_rank() == 0 )
  warp_res = atomicAdd( ctr, g.size() );
```

Solución: *Warp-Aggregated Atomics*

4. Difusión del resultado.

- El thread líder transmite el resultado de `atomicAdd()` a las demás vías del *Warp*.
- Esto se puede hacer mediante una operación de tipo **`shuffle`** en las vías activas.
- Con Grupos Cooperativos se puede transmitir el resultado mediante

```
g.shfl( warp_res, 0 ).
```

- El 0 es el índice del thread líder, lo cual funciona ya que solo los threads activos forman parte del grupo (al haberse creado mediante `coalesced_threads()`).

5. Cálculo del resultado para cada vía.

- En el último paso se calcula la posición de salida de cada vía sumando el valor del contador transmitido para el *Warp* al rango de las vías activas.
- En grupos cooperativos es así:

```
return g.shfl(warp_res, 0) + g.thread_rank();
```

Solución: *Warp-Aggregated Atomics*

Código final resultado.

- Uniendo los fragmentos de código de los pasos 1 a 5 se obtiene la versión completa de la función de incremento, agregada por *Warp*.
- Con los Grupos Cooperativos, el código es conciso y claro.

```
__device__ int atomicAggInc(int *ctr) {
  auto g = coalesced_threads();
  int warp_res;
  if(g.thread_rank() == 0)
    warp_res = atomicAdd(ctr, g.size());
  return g.shfl(warp_res, 0) + g.thread_rank();
}
```

Conclusiones: *Warp-Aggregated Atomics*

- *Warp-Aggregated Atomics* es una técnica útil para mejorar el rendimiento de aplicaciones que realizan muchas operaciones con un número reducido de contadores.
- La mejora de rendimiento puede ser significativa.
- Esta técnica ahora está implementada por defecto en el compilador NVCC y, en muchos casos, la agregación *Warp* se obtiene por defecto sin necesidad de esfuerzo adicional.
- Esta técnica no se limita al *filtrado* ya que puede utilizarse en muchas otras aplicaciones que utilizan operaciones atómicas.