Scan Sky: CUDA

García Prado, Sergio
sergio@garciparedes.me

Calvo Rojo, Adrián
adrian.calvo.rojo@alumnos.uva.es

19 de mayo de 2017

Resumen

En este documento se relatan el conjunto de mejoras aplicadas a un código secuencial base. El modelo de paralelización utilizado para dicha tarea ha sido GP-GPU*(General Purpouse computing on Graphics Processing Units), para lo cual se ha utilizado el framework* CUDA*[1] en su versión para el lenguaje de programación* C*. El código fuente obtenido tras dichas optimizaciones se podrá consultar a través de* `https://github.com/garciparedes/parallel-scan-sky/blob/master/cuda/src/ScanSky_cuda.cu` *[2]*

I. Introducción

En este documento se exponen el conjunto de mejoras realizadas sobre "un código secuencial para contar el número de objetos diferentes que se ven en una imagen o fotografía celeste, en general de espacio profundo, obtenida por un radiotelescopio"[3]

"Las imágenes ya han sido procesadas, discretizando la luminosidad observada en cada punto en 16 diferentes niveles de grises o colores. Los pixels del fondo del espacio, una vez eliminado el ruido, tienen el índice de color 0. Los pixels de la imagen con una luminosidad o color suficientemente parecidos se representan con un mismo valor entre 1 y 15."[3]

"La imagen se carga en una matriz desde un fichero de texto plano. El fichero contiene un número entero en cada línea. Las dos primeras líneas contienen el número de filas y columnas de la imagen. El resto son números entre 0 y 15 con los valores de cada pixel, ordenados por filas."[3]

"Los pixels del mismo índice de color que están juntos, en horizontal o vertical (no en diagonal), se considera que son del mismo objeto. El programa etiqueta cada objeto de la imagen con una etiqueta diferente. Todos los pixels del mismo objeto tendrán la misma etiqueta. Para determinar el número de objetos, al final se cuentan el número de etiquetas diferentes. Los píxeles de índice 0 no se etiquetan."[3]

II. Optimización

Para la mejora de rendimiento en este trabajo se ha utilizado el modelo de programación paralela basado en **SIMD**(una instrucción realizada sobre muchos datos) puesto que el dispositivo físico de soporte utilizado han sido tarjetas gráficas. Tal y como se ha indicado anteriormente se ha utilizado el modelo de *CUDA* desarrollado por *NVIDIA*.

Lo primero es indicar los tamaños de bloque utilizados para la computación en el dispositivo externo: Se han escogido bloque de **16 x 8** tras realizar varias pruebas para encontrar el tamaño óptimo. Puesto que se ha utilizado una estrategia de programación especulativa, también ha sido necesario escoger el nivel de especulación (número de pasos hacia delante a realizar), que siguiendo el mismo procedimiento de experimentación, se ha fijado en **4 streams**.

El modelo de computación *CUDA* describe las secciones de código procesado en el dispositivo como *Kernels*. En este caso se han utilizado **3 Kernels**, referidos a las fases de rellenado de la matriz, propagación de los indices entre las figuras iguales y conteo de figuras distintas. Estos *kernels* se muestran en las figuras 1, 2 y 3 respectivamente.

Para el procesamiento de las matrices se ha decidido dedicar cada hilo a una celda de la misma (puesto que en el modelo de computación con apoyo de GPUs el número de unidades de cómputo generalmente no es un problema). Algo a destacar es la necesidad de "filtrar" el procesamiento para que los hilos sobrantes (debido a la cantidad de grids, que debe ser siempre mayor a los necearios) mediante operadores condicionales en lugar de los bucles iterativos usados en la solución secuencial.

```
__global__ void kernelFillMatrixResult(int *matrixResult) {
    const int ij = (blockIdx.y * blockDim.y + threadIdx.y)*columns_d +
        blockIdx.x * blockDim.x + threadIdx.x;
 if(ij > -1 && ij<rows_d*columns_d){
  if(matrixData_d[ij] !=0){
            matrixResult[ij]=ij;
  } else {
            matrixResult[ij]=-1;
  }
 }
}
```

Figura 1: *kernelFillMatrix()*

```
__global__ void kernelComputationLoop(int *matrixResult,int *matrixResultCopy,
    char *flagCambio_d) {
 const int i = blockIdx.y * blockDim.y + threadIdx.y;
 const int j = blockIdx.x * blockDim.x + threadIdx.x;
 if(i > 0 && i<rows_d-1 &&
  j > 0 && j<columns_d-1 &&
  matrixResult[i*columns_d+j] != -1){
  matrixResult[i*columns_d+j] = matrixResultCopy[i*columns_d+j];
  if((matrixData_d[(i-1)*columns_d+j] == matrixData_d[i*columns_d+j]) &&
   (matrixResult[i*columns_d+j] > matrixResultCopy[(i-1)*columns_d+j]))
  {
   matrixResult[i*columns_d+j] = matrixResultCopy[(i-1)*columns_d+j];
   *flagCambio_d = 1;
  }
  // ...
 }
}
```

Figura 2: *kernelComputationLoop()*

En los dos primeros *kernels* no es necesario realizar sincronizaciones entre bloques o procesos. Sin embargo, en el último se realiza una reducción para el conteo de figuras distintas. La alternativa escogida ha sido la utilización de la función `atomicAdd()`, que a pesar de no ofrecer un rendimiento tan óptimo como la solución manual, simplifica en gran medida la tarea.

```
__global__ void kernelCountFigures(int *matrixResult) {
 const int i = blockIdx.y * blockDim.y + threadIdx.y;
  const int j = blockIdx.x * blockDim.x + threadIdx.x;
 if(i > 0 && i<rows_d-1 &&
  j > 0 && j<columns_d-1 &&
   matrixResult[i*columns_d+j] == i*columns_d+j) {
    atomicAdd(&numBlocks_d, 1);
 }
}
```

Figura 3: *kernelCountFigures()*

Una vez descritos los *kernels* implementados se describen las estructuras de datos en que se ha apoyado el cómputo, estas se muestran en la figura 4. En algunos casos se ha utilizado la zona de memoria constante, que reduce el tiempo de acceso a la misma, mientras que en otros el almacenamiento se ha apoyado en la memoria compartida del dispositivo.

Algo a destacar es el "casteo" de la matriz de datos del tipo `int` al tipo `char` lo cual reduce en 4 el tamaño de la misma. Esta operación se realiza en la CPU porque la redución posterior del tiempo de transferencia lo compensa. En esta parte sucede un suceso curioso para el cual no se ha encontrado explicación. La reserva de memoria se hace a partir de la función `cudaMallocPitch()`, que se supone que alinea la memoria para mejorar la eficiencia, por lo que la documentación indica que para la transferencia de datos entre la *CPU* y el *Dispositivo Externo* es necesario utilizar `cudaMemcpy2DAsync()` para ser consistente con el alineamiento. Sin embargo, el uso de esta función conlleva un resultado erróneo, mientras que la transferencia mediante `cudaMemcpyAsync()` si que proporciona un funcionamiento válido.

El segundo factor a destacar (derivado del uso de procesamiento *multi-streaming*) es la fusión de la `matrixResult` y `matrixResultCopy` en una nueva matriz de gran tamaño con lo que se consigue que estas estén colocadas de ma-

nera contigua estrictamente en memoria. A pesar de ello esta no ha sido la razón principal, sino la de poder llevar a cabo fases de **programación especulativa** en cuanto al número de iteraciones en `kernelComputationLoop()` (para lo cual es necesario poseer tantas matrices como niveles de especulación se desee llevar a cabo).

```
gpuErrorCheck(cudaMalloc(&flagCambio_d, sizeof(char) * nStreams));
gpuErrorCheck(cudaMallocPitch(&matrixResult_d, &pitch, rows*sizeof(int), columns * nStreams));
gpuErrorCheck(cudaMallocPitch(&matrixDataChar_d, &pitch3, rows*sizeof(char), columns));
gpuErrorCheck(cudaMemcpyToSymbolAsync(rows_d,&rows, sizeof(int),0,cudaMemcpyHostToDevice));
gpuErrorCheck(cudaMemcpyToSymbolAsync(columns_d,&columns, sizeof(int),0,cudaMemcpyHostToDevice));
gpuErrorCheck(cudaMemcpyToSymbolAsync(matrixData_d,&matrixDataChar_d, sizeof(char *)));
matrixDataChar= (char *)malloc( rows*(columns) * sizeof(char) );
for(i = 0; i < rows * columns; i++){
    matrixDataChar[i] = matrixData[i];
}
gpuErrorCheck(cudaMemcpyAsync(matrixDataChar_d,matrixDataChar,
    sizeof(char) * rows * columns,cudaMemcpyHostToDevice));
```

Figura 4: *Reserva de memoria para las Estructuras de Datos*

A continuación se describen las distintas fases de la implementación mediante las llamadas a los *kernels*. En primer lugar se realiza una llamada a `kernelFillMatrix()` para que rellene el segmento correspondiente a la anterior `matrixResult` que ahora se representa en la primera parte de `matrixResult_d`.

```
kernelFillMatrixResult<<<gridShapeGpu, bloqShapeGpu>>>(
    &matrixResult_d[rows * columns * 0]
);
gpuErrorCheck(cudaPeekAtLastError());
```

Figura 5: *Fase de Rellenado*

Una vez hecho esto comienza la sección de cómputo, dominada por la variable `flagCambio`. Debido a la estrategia de *Programación Especulativa*, primeramente se realiza un lanzamiento secuencial para los distintos niveles de especulación sobre el `stream[0]` para que estos permanezcan sincronizados durante el resto del cómputo, ya que su nuevo lanzamiento no se llevará a cabo hasta haberse completado el anterior. Esto se extrapola a su lanzamiento en diferentes streams y la variable `flagCambio` en el bucle `t`.

```
gpuErrorCheck(cudaMemsetAsync(flagCambio_d, 0, sizeof(char) * 4, stream[0]));
for (i = 0; i < nStreams; i++){
    kernelComputationLoop<<<gridShapeGpu, bloqShapeGpu,0,stream[0]>>>(
        &matrixResult_d[rows * columns * ((i+1)%nStreams)],
        &matrixResult_d[rows * columns * i],
        &flagCambio_d[i]
    );
}
gpuErrorCheck(cudaPeekAtLastError());
int s = -1;
for(t=0; flagCambio != 0; t++){
    flagCambio = 0;
    s = t % nStreams;
    gpuErrorCheck(cudaMemcpyAsync(&flagCambio,&flagCambio_d[s], sizeof(char),
        cudaMemcpyDeviceToHost,stream[s]));
    gpuErrorCheck(cudaMemcpyAsync(&flagCambio_d[s],&zero, sizeof(char),
        cudaMemcpyHostToDevice,stream[s]));
    kernelComputationLoop<<<gridShapeGpu, bloqShapeGpu,0,stream[s]>>>(
        &matrixResult_d[rows * columns * ((s+1)%nStreams)],
        &matrixResult_d[rows * columns * s],
        &flagCambio_d[s]
    );
}
```

Figura 6: *Bucle Principal de Cómputo*

Nótese que esta estrategia no reduce el número de iteraciones necesarias para la resolución del problema, sino que solapa fases de transferencia de datos y computación en el dispositivo externo.

```
numBlocks = 0;
gpuErrorCheck(cudaMemcpyToSymbolAsync(numBlocks_d,&numBlocks,
    sizeof(int),0,cudaMemcpyHostToDevice));
kernelCountFigures<<<gridShapeGpu, bloqShapeGpu>>>(
    &matrixResult_d[rows * columns * ((t + 1) % nStreams)]
);
gpuErrorCheck(cudaPeekAtLastError());
gpuErrorCheck(cudaMemcpyFromSymbolAsync(&numBlocks, numBlocks_d,
    sizeof(int), 0, cudaMemcpyDeviceToHost));
```

Figura 7: *Conteo de figuras en la matriz*

El último paso es el conteo de figuras en la matriz de resultados (la parte correspondiente de `matrixResult_d` respecto de la iteración la cuál finalizó el cómputo). Esto se ilustra en la figura 7 en la cual se realiza la llamada al *kernel* `kernelCountFigures()`. Una vez finalizada dicha operación, se devuelve a la *CPU* el valor deseado y el programa finaliza tras el pertinente vaciado de memoria, tanto en la *CPU* como en el *dispositivo externo*.

```
#define gpuErrorCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort = true) {
    if (code != cudaSuccess) {
  fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
  if (abort) { exit(code); }
    }
}
```

Figura 8: *Estrategia de validación de Errores*

Por último, se ha añadido una parte destinada al control de errores producidos en el dispositivo externo, la cual termina la ejecucción e imprime en salida estándar el mensaje de error, junto con la línea y el fichero donde se produjeron. Esta implementación ha sido extraida de un ejemplo de código al cual se puede acceder a partir de: `http://www.orangeowlsolutions.com/archives/613` [4]

III. Conclusiones y Propuestas Futuras

El uso de dispositivos externos que siguen el paradigma **SIMD** para accelerar regiones inerentemente paralelas como operaciones sobre estructuras de datos matriciales proporciona una gran ventaja respecto de implementaciones secuenciales. Sin embargo, es muy importante la correcta gestión de la memoria del dispositivo para que siempre tenga trabajo para realizar.

En cuanto a propuestas futuras de trabajo, se puede tratar de mejorar los accesos a memoria utilizando almacenando en el dispositivo inicialmente (mediante `__device__`) el mayor número de variables posible. Otra de las propuestas futuras es la extensión de dicha solución a varios dispositivos externos (puesto que actualmente tan solo es posible su ejecución en uno de ellos).

Referencias

[1] NVIDIA CUDA. `https://developer.nvidia.com/cuda-zone`.

[2] García Prado, S., and Calvo Rojo, A. Parallel Scan Sky. `https://github.com/garciparedes/parallel-scan-sky`.

[3] González Escribano, A., Moretón Fernández, A., and Rodríguez Gutierrez, E. Computación paralela, 2016/17.

[4] Solutions, O. O. Tricks and Tips: cudaMallocPitch and cudaMemcpy2D. `http://www.orangeowlsolutions.com/archives/613`.