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

[TODO]

```
__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: *[TODO]*

```
__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: *[TODO]*

```
__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: *[TODO]*

```
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: *[TODO]*

```
kernelFillMatrixResult<<<gridShapeGpu, bloqShapeGpu>>>(
    &matrixResult_d[rows * columns * (nStreams-1)]
);
gpuErrorCheck(cudaPeekAtLastError());
```

Figura 5: *[TODO]*

```
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],
        &matrixResult_d[rows * columns * ((i - 1 + nStreams) % nStreams)],
        &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],
        &matrixResult_d[rows * columns * ((s - 1 + nStreams) % nStreams)],
        &flagCambio_d[s]
    );
}
```

Figura 6: *[TODO]*

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

Figura 7: *[TODO]*

```
#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: *[TODO]*

III. Conclusiones y Propuestas Futuras

[TODO]

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.