高性能 PA3

计83 李天勤 2018080106

实验目的

The goal of this experiment is to become familiar with CUDA programming by implementing a GPU-accelerated Floyd-Warshall algorithm, and utilizing a common optimization method known as blocking.

实验实现

I wanted to test the performance of the Floyd Warshall algorithm on the CPU, so I used `cudaMemcpy` to copy data from device to host memory allocated by malloc, and ran the algorithm on the host.

```
// copy device graph to host
    cudaMemcpy(graph_host, graph, bytes, cudaMemcpyDeviceToHost);
    // cpu test
    auto beg = std::chrono::high_resolution_clock::now();
    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                graph_host[i * n + j] = MIN(graph_host[i * n + j], graph_host[i
* n + k] + graph_host[k * n + j]);
    auto end = std::chrono::high_resolution_clock::now();
    double t = std::chrono::duration_cast<std::chrono::duration<double>>(end -
beg).count() * 1000;
    printf("floyd on cpu = %f ms", t); // ms
```

running the first test case, `srun -N 1 ./benchmark 1000` resulted in

```
floyd on cpu = 951.285802 ms
floyd on cpu = 952.483183 ms
floyd on cpu = 951.310510 ms
```

The GPU brute force method given in `apspRf.cu`

```
for (int k = 0; k < n; k++) {
    dim3 thr(BLOCK_SIZE, BLOCK_SIZE);
    dim3 blk((n - 1) / BLOCK_SIZE + 1, (n - 1) / BLOCK_SIZE + 1);
    naive_kernel<<<blk, thr>>>(n, k, graph);
}
```

contained a grid of $(n-1)/32+1 \times (n-1)/32+1$ blocks, each block containing 32×32 threads. In a case of $n = 100$, there will be a thread block size of 32×32, and there will be 4×4 blocks. The `kernel` will run on these different CUDA threads, each thread running a simple matrix operations.

Compared to the CPU, the GPU had already made the program about 60x faster for the test case $n = 1000$. This is perhaps the simplest implementation, as it just creates threads containing single iterations of the innermost loop. Since a GPU can efficiently schedule a large number of threads, this approach works relatively well compared to the CPU. However this method doesn't utilize cache efficiently and thus saturates the global memory bus.

In order to make the algorithm utilize GPU more efficiently, I implemented a Blocked Floyd Warshall APSP. This method is derived from the article "A Multi-Stage CUDA Kernel for Floyd-Warshall" written by Ben Lund and Justin W. Smith.

The method proposed in the lab guide is an implementation of Blocked Floyd-Warshall by Katz and Kider.

There are 3 stages, 3 kernels, 32 x 32 tiles, 32 tasks for each data element during each stage

1. The first kernel processes 32 tasks for each data element in single tile on the diagonal of the adjacency matrix. Tasks in this independent block depend only on other tasks in the independent block, or on tasks that were completed in previous stage.
2. The second kernel calculates 32 tasks for each data element in each tile aligned with the independent block in either the i or j direction. These are known as singly dependent blocks. And tasks in the singly dependent blocks have one data dependency within the block and one dependency that has already been calculated in the independent blocks
3. The third kernel processes 32 tasks for each of the remaining tiles. These are the doubly dependent blocks, and each of these blocks depends entirely on two singly dependent blocks.

This method above utilizes the cache, by utilizing the shared memory associated with each processer, performing several tasks for each data element that is transferred, thus reducing the total data that has to be sent through the bus.

I tried implementing `#pragma unroll` to reduce instruction count, in attempt to make lengthy for loops quicker but in the following testcases, there was little to no impact to performance when using it to unroll for loops.

实验结果

Benchmarks using Brute Method `apspRef.cu`

```
n = 1000
Time: 14.999313 ms

n = 2500
Time: 377.044251 ms

n = 5000
Time: 2972.033733 ms

n = 7500
Time: 10012.611050 ms

n = 10000
Time: 22629.584993 ms
```

加速比例

```
n = 1000
```

```
Time: 2.964226 ms
加速: 14.999313/2.964226 = 5.06x

n = 2500
Time: 22.788418 ms
加速: 377.044251/22.788418 = 16.55x

n = 5000
Time: 158.712036 ms
加速: 2972.033733/158.712036 = 18.73x

n = 7500
Time: 516.604813 ms
加速: 10012.611050/516.604813 = 19.38x

n = 10000
Time: 1199.500483 ms
加速: 22629.584993/1199.500483 = 18.87x
```

实验分析、参考材料

The GPU provides much higher instruction output and memory bandwidth compared to the CPU, allowing us to run many times more computations in parallel compared to a CPU. From the results we can see, that even the most basic implementation of GPU is considerably more powerful than the CPU when it comes to data processing such as computing matrix operations.

参考材料

https://arxiv.org/pdf/1001.4108.pdf

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#from-graphics-processing-to-general-purpose-parallel-computing

https://stackoverflow.com/questions/16119943/how-and-when-should-i-use-pitched-pointer-with-the-cuda-api

from docs.nvidia

```
cudaMalloc(void **devPtr, size_t count); // allocates memory of size count in device memory and updates the deevice pointer devPtr to the allocated memory
cudaDeviceSynchronize(); // blocks until device has completed all preceding requested tasks, returns cudaSuccess
cudaMemcpy(void * dst, const void* src, size_t count, cudaMemcpyDefault); // copies count bytes from memory area pointed by src to memory area pointed by dst
cudaMemcpy2D(void * dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, cudaMemcpyDeviceToHost); //copies a matrix (height rows of width bytes each) from memory area pointed to by src to the memory area pointed to by dst
```