EECS-4750 Assignment 3

2D kernels, Matrices, Shared Memory, Constant Memory

Xinghua Sun - xs2445

Programming Problems

Before we look into the convolution problem, some review of the memory model is necessary.

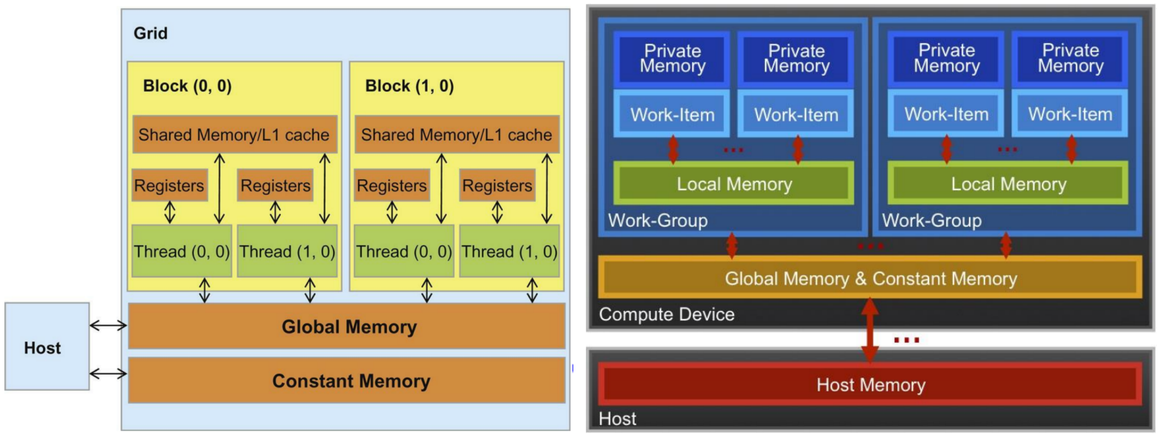

Fig 1 Memory model of CUDA and OpenCL

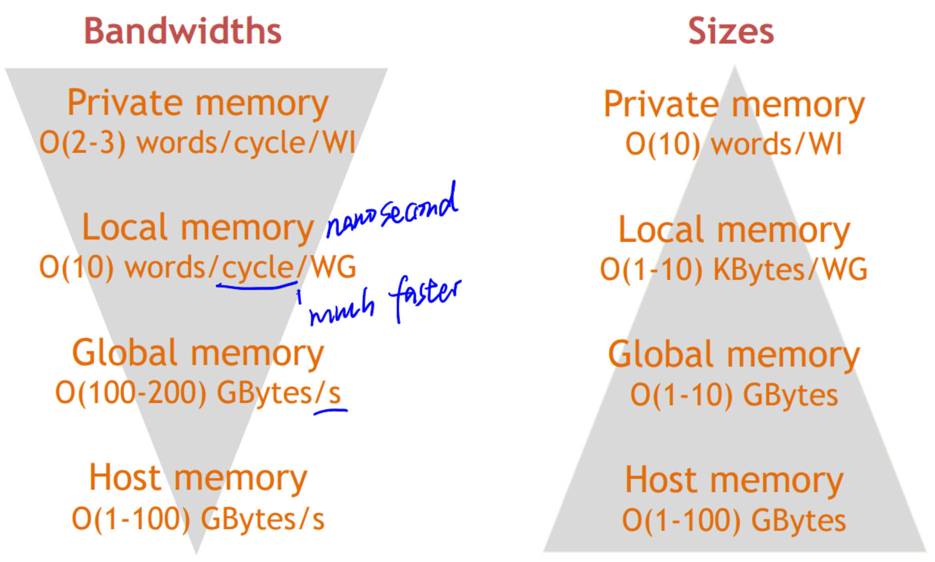

Fig 2 Memory Hierarchy (OpenCL memory model)

As the Fig 1 and Fig 2 shows, as the memory has a larger bandwidth, the size of it shunks. So we need to consider which part of the memory should be used for which part of the operation. For the good of performance, which is the speed of the operation, we need to make more use of faster memories. Meanwhile not to use too much of it, in case the memory will be exceeded which will slow down the program or even cause a bug.

For the convolutional problem, the size of the input matrix and the mask is not fixed. But usually the input matrix is very large and the mask is rather small. That gives us some ideas of choosing memory.

1. Convolution using global memory

A very intuitive way for the convolution is to use the largest memory for every operation, and do not think of the cost of time. That means we only use the global memory so that we don't need to consider whether the memory has enough space for the data.

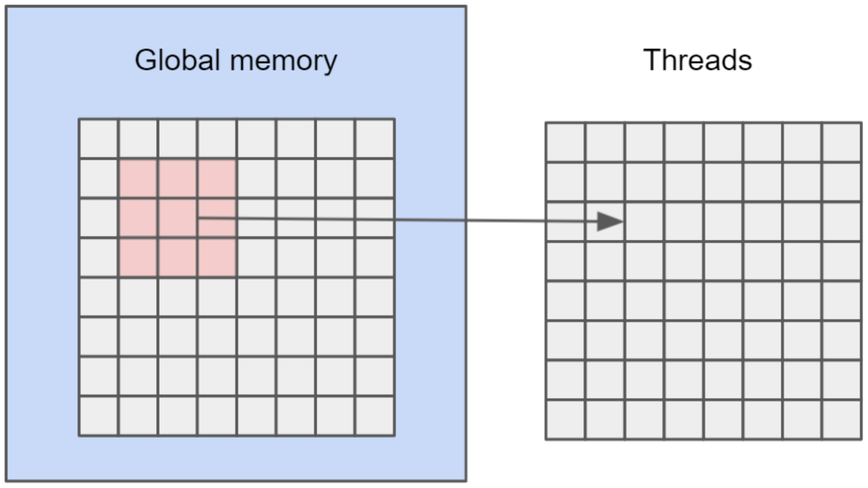

Fig 3 Only use global memory

As Fig 3 shows, each thread will calculate for one output element of the result matrix. Each thread needs to access the global memory for 18 times. The global memory has a long access latency, which will cause a significant time delay.

2. Convolution using shared memory

In the global memory method, each element of the input matrix will be accessed multiple times. If we can put the input matrix and the mask into the shared memory, then we can avoid using too much global memory, which can significantly lower the time latency. However, because of the limited size of the shared memory, we need to figure out a method to put the input matrix into shared memory.

As Fig 4 shows, a method called tilling is used to make input matrix tiles, so that we can copy it into shared memory. This method only accesses the input matrix once to avoid the long access latency.

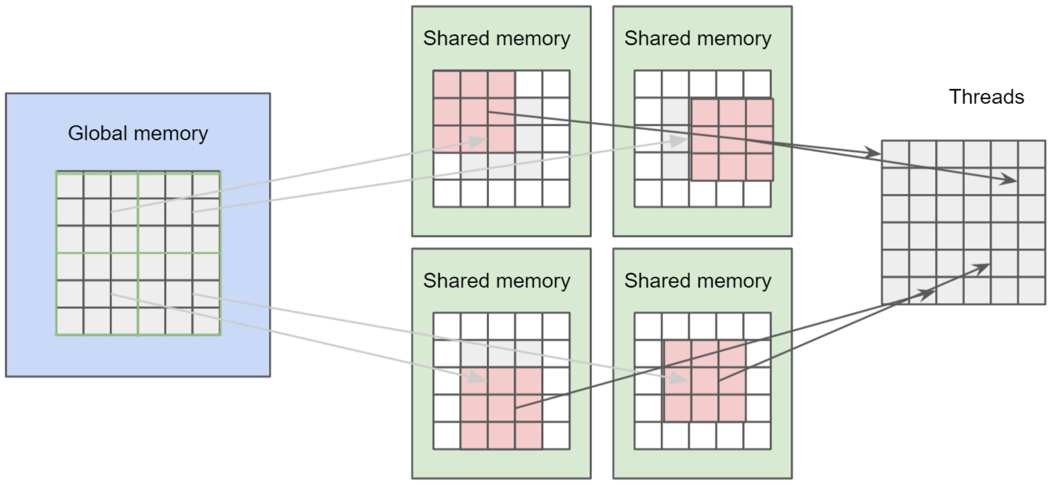

Fig 4 Use shared memory

3. Shared memory and constant memory

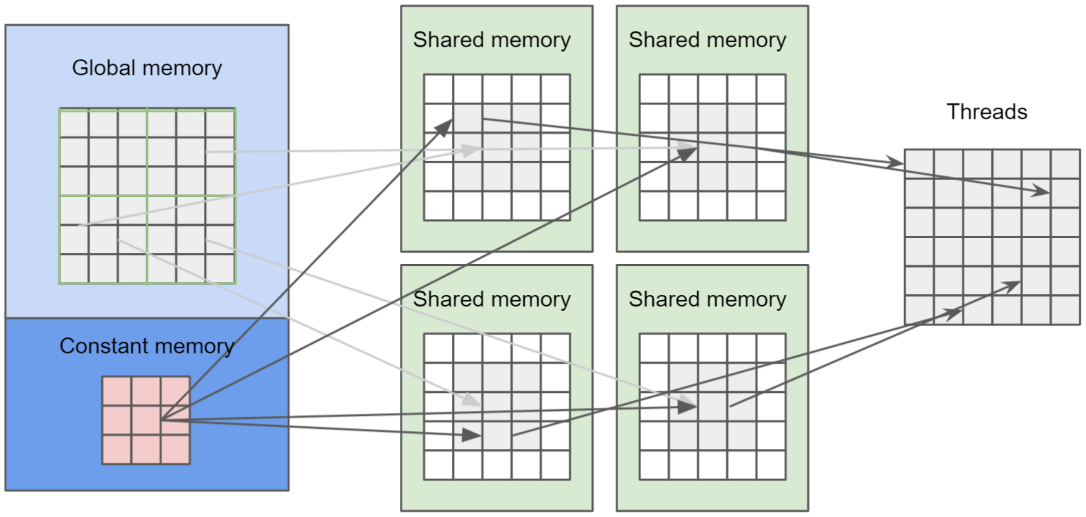

Fig 5 Copy mask to constant memory for caching

The mask is not constant and small, so we can put it into the constant memory to improve the performance. As Fig 6 shows, the modern processors usually use a technique called cache for constantly accessed data to improve performance. Declare a constant variable is telling the device to aggressively cache the variable nito L1 cache, which will further reduce the access latency.

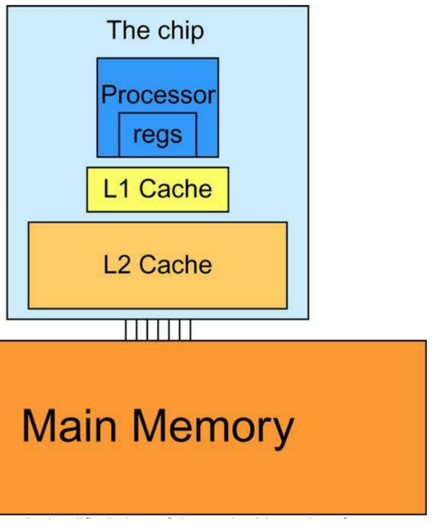

Fig 6 A simplified view of the cache hierarchy of modern processors

4. Comparison of the performance

The performance of each method implemented in PyCUDA and PyOpenCL is illustrated below:

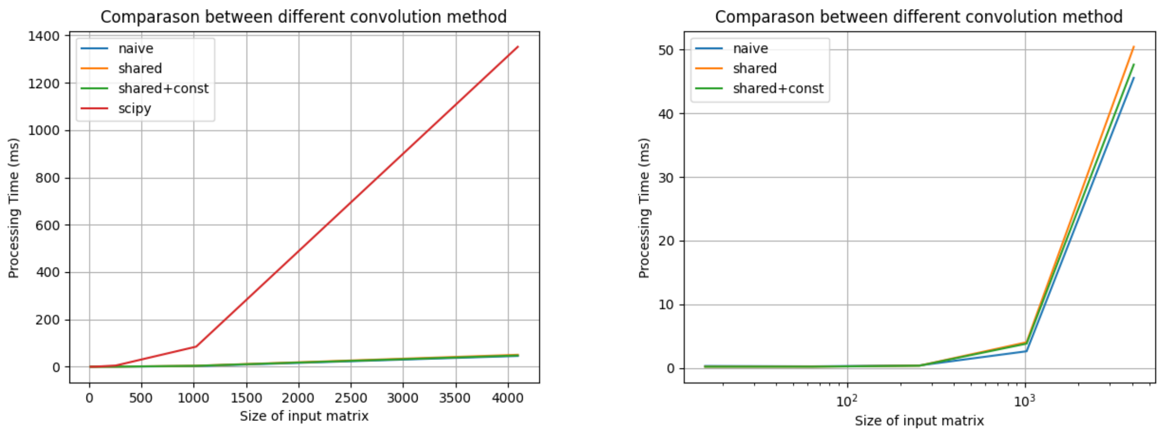

Fig 7 PyCUDA implementation

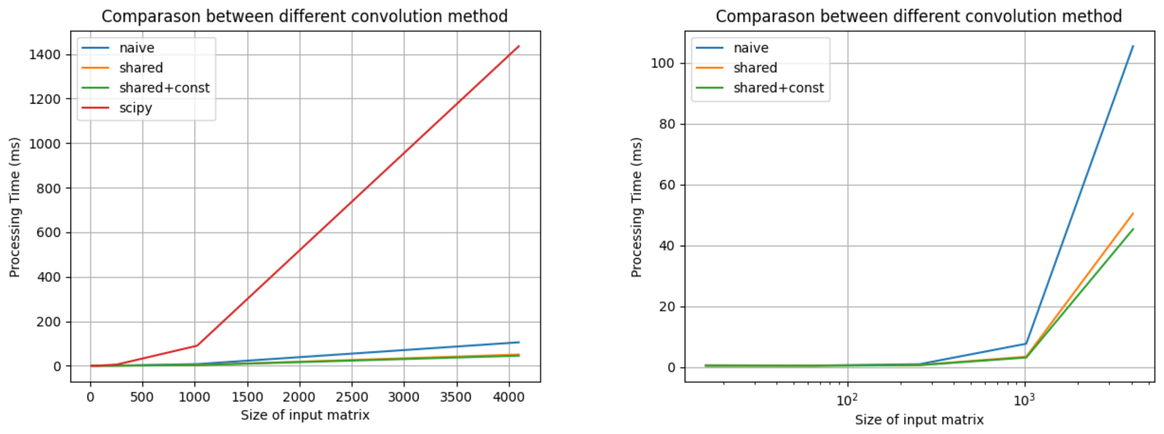

Fig 8 PyOpenCL implementation

As size of the input matrix increases, the parallel implementations have a significant improvement on the processing time. The PyOpenCL implementation just like the methods discussed above, the shared memory method is faster than the naive method, and the shared and constant memory method is faster than only using the shared memory method. For the PyCUDA implementation, three methods do not have big differences. But compared to PyOpenCL implementation, the naive method in PyCUDA already meets the fastest performance and even slightly faster than other methods. That's probably because of some low-level optimization by CUDA.

Theory Problems

1. Compare the recorded times against the serial implementation for all the above cases. Which approach is faster in PyCuda? Which approach is faster in PyOpenCL? Why is that particular method better than the other?

In the PyCUDA implementation, the naive method using global memory is the fastest. In the PyOpenCL implementation, the shared and constant memory method is the fastest.

As three methods discussed above, the shared memory has a larger bandwidth, and the constant memory is actually copying the data to the cache, which has an even faster access speed. That's why in the PyOpenCL implementation, copying input matrix to shared memory and mask to constant memory has a better performance.

However, PyCUDA implementation is different. For the second and third methods, shared memory plus constant memory is still faster than only using shared memory. However, the naive method using only global memory is even faster than those two improved methods. That's probably because of the low-level optimization by CUDA.

2. Can this approach be scaled for very big kernel functions? In both cases explain why?

The answer is no, because it will exceed the shared memory.

The reason why we need to tile the input matrix is because of the limited size of the shared memory. It’s the same reason we only put the mask into constant memory. Also to make the convolution as a whole, we used a _synthread() function to wait for all the elements to be loaded into the shared memory.

If the matrix is too large, then the shared memory, either the memory will be exceeded and cause a bug, or the synchronization process will never pass which will also cause a bug. A possible solution for this problem is to tile the matrix in the host or in the global memory (only if the global memory is large enough). Do the convolution for each tile and finally assemble all the results as the final result.

3. Explain what is row-major order and column-major order for storing multidimensional arrays? Does it matter how data is stored?

In random access memory (C/C++ language) the data is stored serial in the memory. In the kernel function, when we are dealing with some matrices whose dimension is larger than 1, we need to firstly localize the element before any operations. Row-major order and column-major order are two conventions of indexing or storing arrays in linear storage.

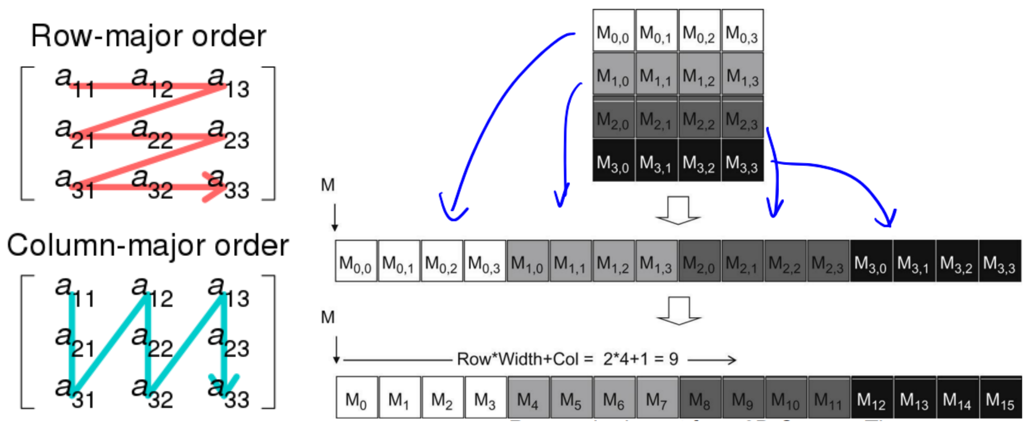

Fig 9 The row-major order and column-major order (the right figure is PyCUDA implementation which is row-major order)

In PyCUDA, arrays follow the row-major order, and in PyOpenCL, arrays follow the column-major order.

It doesn't matter how the arrays are stored, the data will not be lost. But it matters how we localize it.

4. Consider the following kernel for finding the sum of two matrics:

```
__global__
void matrix_sum(float *A, float *B, float *C,
  int m, int n) {
  // assume 2D grid and block dimensions
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  if (x < m && y < n) {
    int ij = x + y*m; // column-major order
    C[ij] = A[ij] + B[ij];
  }
}
```

Now consider the call to the kernel:

```
// optimization: copy data outside of the loop
cudaMemcpy(dA,...);
cudaMemcpy(dB,...);
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j) {
    int ij = i + j*n; // column-major order
    matrix_sum<<<1,1>>>(dA+ij, dB+ij, dC+ij, 1,1);
  }
cudaMemcpy(hC,dC,...);
```

Will this code work? If yes - is this efficient, why or why not. If not, what can you do to improve the efficiency? If not, what is the issue with this code and how would you fix the issue?

The code will work but is inefficient.

To implement parallel computation, the kernel is fine, the problem is in the host code.

In the loop of the host code, `dA+ij, dB+ij, dC+ij` are three pointers of the ij-th element in each array. Because every time the grid size and block size is 1, so in each iteration only one thread will be used, and the ij-th element will be calculated in each iteration and stored into the ij-th position of dC.

Although it can work, it's inefficient because it's actually a serial implementation. The advantage of parallel programming is that multiple threads can work at the same time. So we can get the result of multiple threads with the same kernel of different data processed. The processing speed of streaming processors in the GPU is a disadvantage compared to the CPU. So this implementation will probably be slower than the serial implementation on CPU.

To improve it, only the host code needs to be modified.

```
// optimization: copy data outside of the loop
cudaMemcpy(dA,...);
cudaMemcpy(dB,...);
GridDim = (ceil(n/256.), ceil(n/256.), 1)
BlockDim = (256,256,1)
matrix_sum<<<GridDim, BlockDim>>>(dA, dB, dC, n, n)
cudaMemcpy(hC,dC,...);
```

Reference

kirk, Daivid B., and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, 2017.

McIntosh-Smith, Simon, and Tom Deankin. *Hands On OpenCL*. https://handsonopencl.github.io/.

"PyCUDA Documentation." *pycuda 2021.1 documentation*, https://documen.tician.de/pycuda/index.html.

"PyOpenCL Documentation." *pyopencl 2021.2.6 documentation*, https://documen.tician.de/pyopencl/index.html.

"Wikipedia." *Row- and column-major order*

, https://en.wikipedia.org/wiki/Row-_and_column-major_order.