Matrix Multiplication

OpenCL Tutorial
Part 2.2

Guillermo Marcus
guillermo.marcus@gmail.com

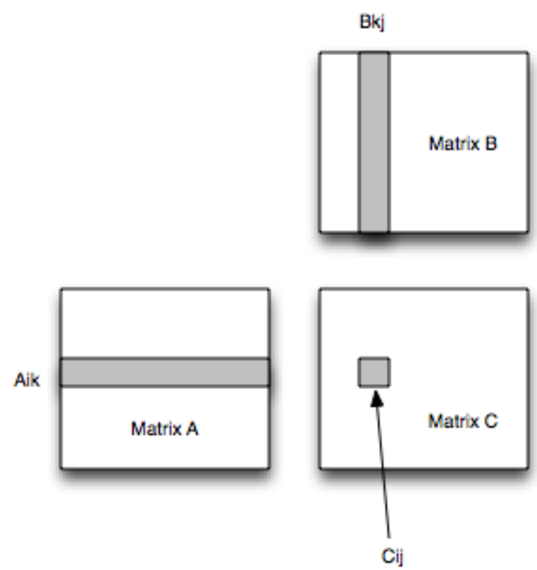
Matrix Multiplication

$$C = A * B$$

$$C[c_{width} \times c_{height}] = A[a_w \times a_h] * B[b_w \times b_h]$$

$$c_{ij} = \sum_k a_{ik} * b_{kj}$$

We use square Matrices in this example



Matrix Multiplication - CPU

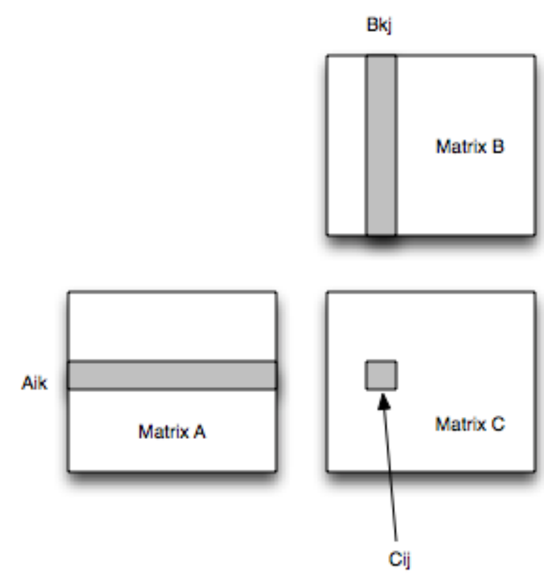
```
void matmul( int side, int *A, int *B, int *C ) {
	int i, j, k;

	for(j=0; j < side; j++)
		for(i=0; i< side; i++) {
			int c_idx = j*side + i;
			C[c_idx] = 0;
			for(k=0;k< side; k++) {
				int a_idx = j*side + k;
				int b_idx = k*side + i;
				C[c_idx] += A[a_idx] * B[b_idx];
			}
		}
	return;
}
```

OpenCL - Host code

```
// Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, array_size * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, array_size * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, array_size * sizeof(int) );

// Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, array_size * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, array_size * sizeof(int), B );

// Create kernel specification (ND range)
        NDRange global(array_size);
        NDRange local(1);

// Set kernel arguments
        kernel.setArg(0, side);
        kernel.setArg(1, bufA);
        kernel.setArg(2, bufB);
        kernel.setArg(3, bufC);

// Run kernel
//              queue.enqueueNDRangeKernel( kernel, NullRange, global, local );
                queue.enqueueNDRangeKernel( kernel, NullRange, global, NullRange ); // MUCH better!

// Copy result buffer from device
                queue.enqueueReadBuffer( bufC, CL_TRUE, 0, array_size * sizeof(int), C );
```

OpenCL - Kernel code

```
__kernel void matrixMul(__const int side, __global const int *A, __global
const int *B, __global int *C) {
	int id = get_global_id(0);

	int j = (id / side);
	int i = (id % side);

	int a_idx;
	int b_idx;

	__private int sum = 0;
	for(int k=0; k< side; k++) {
		a_idx = j * side + k;		// A[i][k]
		b_idx = k * side + i;		// B[k][j]
		sum += A[a_idx] * B[b_idx];
	}
	C[id] = sum;				// C[i][j]
}
```

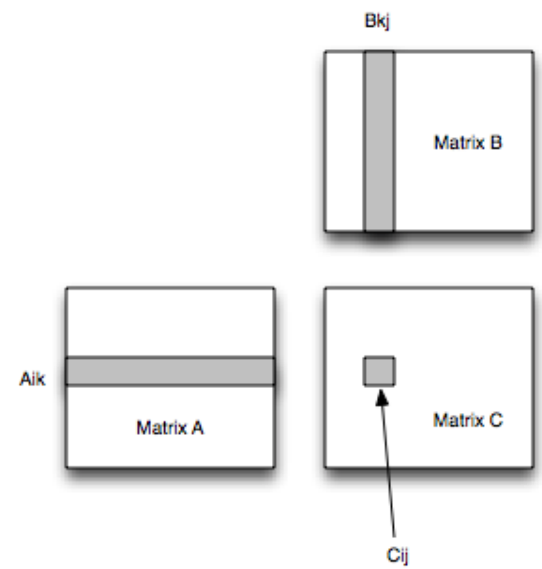


Using 2D coordinates - Host

```
// Create kernel specification (ND range)
    NDRange global(side,side);
    NDRange local(1,1);

// Set kernel arguments
    kernel.setArg(0, side);
    kernel.setArg(1, bufA);
    kernel.setArg(2, bufB);
    kernel.setArg(3, bufC);

// Run kernel
    queue.enqueueNDRangeKernel( kernel, NullRange, global, local );
```

Using 2D coordinates - Kernel

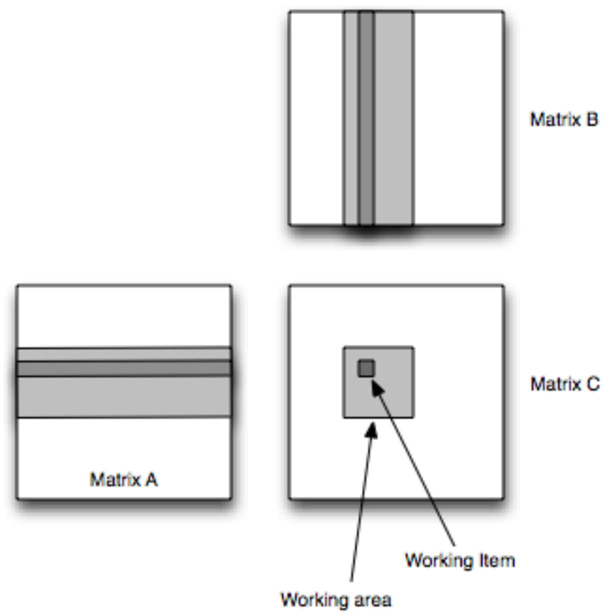
```
__kernel void matrixMul(__const int side, __global const int *A, __global
const int *B, __global int *C) {

    int i = get_global_id(0);
    int j = get_global_id(1);

    int id = j*side + i;

    int a_idx;
    int b_idx;

    __private int sum = 0;
    for(int k=0; k< side; k++) {
        a_idx = j * side + k;      // A[j][k]
        b_idx = k * side + i;      // B[k][i]
        sum += A[a_idx] * B[b_idx];
    }
    C[id] = sum;                   // C[j][i]
}
```



Task 3: Using local memory

If we read multiple rows and columns into local memory, we can reuse the values to compute a subsection of the target matrix.

We gain, that we read the values from main memory only once.



Task 3 - What to do?

Now we need a rectangular definition of threads
-> Change this in the host code

Then, in the GPU kernel, we need...

- To define the local memory to use
- To prefetch the data into local memory
- Compute from local memory

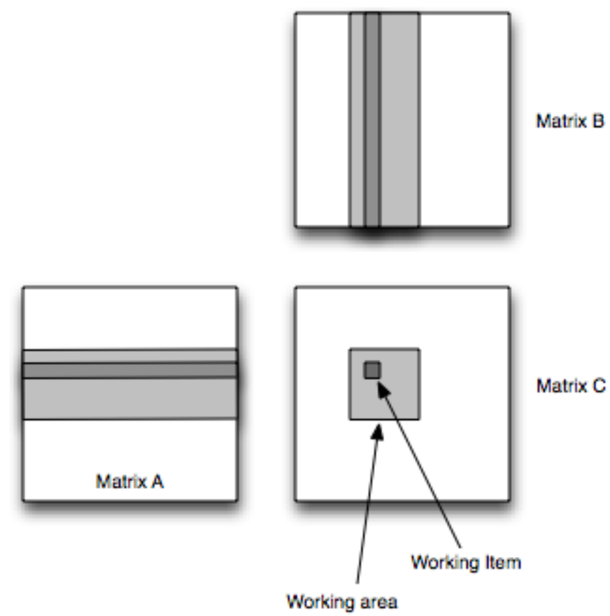
Finally, write into Main memory the result.

Task 3 - Solution - Host code

```
// Create kernel specification (ND range)
    int groups = (side / 8) + ((side % 8 == 0) ? 0 : 1);
    NDRange global(groups*8,groups*8);
    NDRange local(8,8);

// Set kernel arguments
    kernel.setArg(0, side);
    kernel.setArg(1, bufA);
    kernel.setArg(2, bufB);
    kernel.setArg(3, bufC);
    kernel.setArg(4, cl::__local(8*side*sizeof(int)) );
    kernel.setArg(5, cl::__local(8*side*sizeof(int)) );

// Run kernel
    queue.enqueueNDRangeKernel( kernel, NullRange, global, local );
```

Task 3 - Solution - GPU Kernel

```
__kernel void matrixMul2(__const int side, __global const int *A, __global const int *B, __global int *C, __local int *lA, __local int
*lB) {

        int i = get_global_id(0);
        int j = get_global_id(1);

        int id = i*side + j;

        int li = get_local_id(0);
        int lj = get_local_id(1);

        // prefetch using local memory
        for(int k=0; k< side; k+=8) {
                if ((i < side) && (lj+k < side))
                        lA[ li * side + lj + k] = A[ i * side + lj + k];
                if ((j < side) && (li+k < side))
                        lB[ (li + k)*8 + lj ] = B[ (li + k) * side + j ];
        }

        read_mem_fence( CLK_LOCAL_MEM_FENCE );

        if ((i >= side) || (j >= side))
                return;

        int a_idx;
        int b_idx;

        __private int sum = 0;
        for(int k=0; k< side; k++) {
                a_idx = li * side + k; // A[i][k]
                b_idx = k * 8  + lj;     // B[k][j]
                sum += lA[a_idx] * lB[b_idx];
        }
        C[id] = sum;                            // C[i][j]
}
```



Task 4 - Tiling

Last version has the limitation that we need the rows / columns to fit into the local memory. This limits the target size of the matrix.

We can solve this by prefetching only a part of the row / columns, that is, dividing the data in tiles. We process a tile after the other, adding partial results.

This has advantages in size and also in performance!

Task 4 - Solution

```
__kernel void matrixMul4(__const int side, __global
const float *A, __global const float *B,
__global float *C, __local float *lA, __local float *lB) {

int i = get_global_id(0);
int j = get_global_id(1);

int li = get_local_id(0);
int lj = get_local_id(1);

unsigned long id = i*side + j;

// this is initialized only once. We get one per thread
__private float sum = 0.f;
__private float c = 0.f;
__private float y=0.f, t=0.f;

// iterate over the tiles
for(int offset=0; offset< side; offset += 8) {
	// prefetch 8x8 using local memory
	if ((i < side) && (lj+offset < side))
		lA[ li*8 + lj ] = A[ i * side + lj + offset];
	else
		lA[ li*8 + lj ] = 0.f;
if ((j < side) && (li+offset < side))
	lB[ li*8 + lj ] = B[ (li + offset) * side + j ];
else
	lB[ li*8 + lj ] = 0.f;
barrier( CLK_LOCAL_MEM_FENCE );

if ((i < side) && (j < side)) {
	int a_idx;
	int b_idx;
	for(int k=0; k< 8; k++) {
		a_idx = li*8 + k;! // A[i][k]
		b_idx = k*8 + lj;! // B[k][j]
	// Kahan algorithm for
		sum += lA[a_idx] * lB[b_idx];
		y = ( lA[a_idx] * lB[b_idx] ) - c;
		t = sum + y;
		c = ( t - sum ) - y;
		sum = t;
	}
}
barrier( CLK_LOCAL_MEM_FENCE ); // sync all
threads

}

// we write only after we finalize all tiles
if ((i < side) && (j < side))
	C[id] = sum;		// C[i][j]
}
```