OpenCL

Tópicos em Arquiteturas Paralelas

Peter Frank Perroni

November 25, 2015

The Device

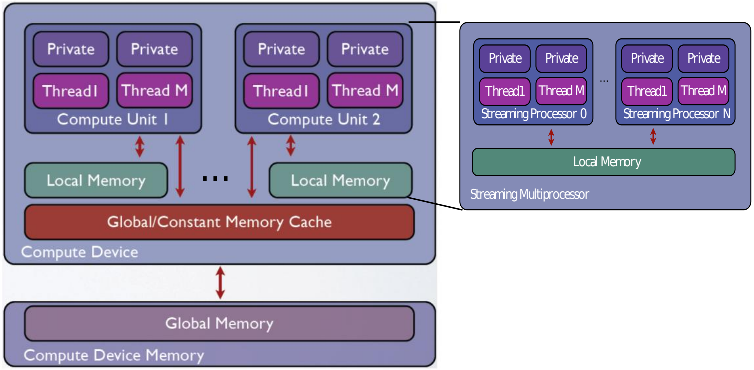


Figure : OpenCL hierarchy.

SM

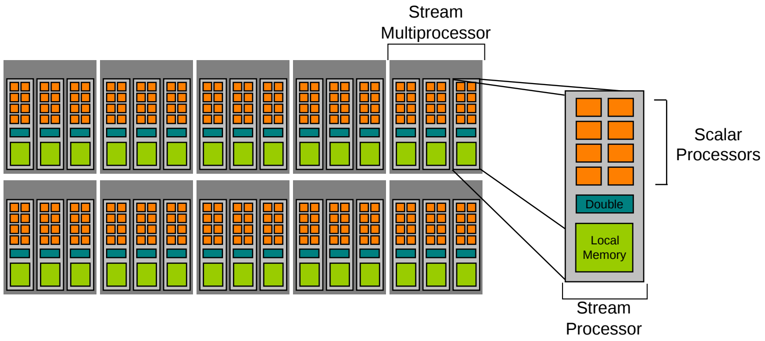
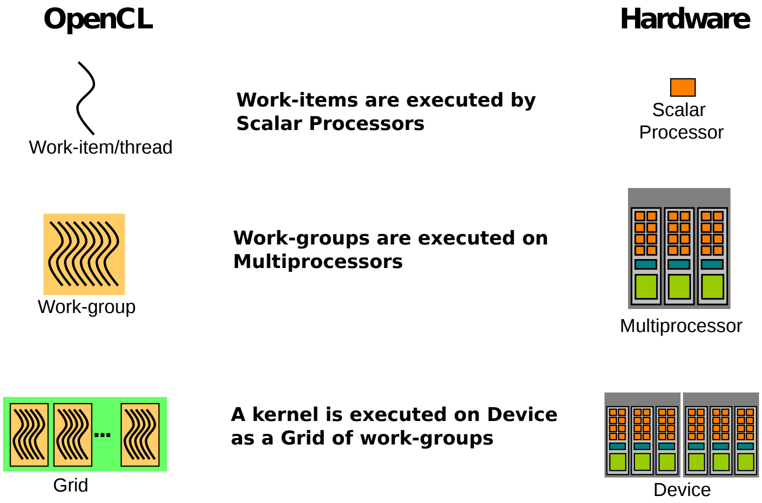


Figure : Stream Multiprocessors.



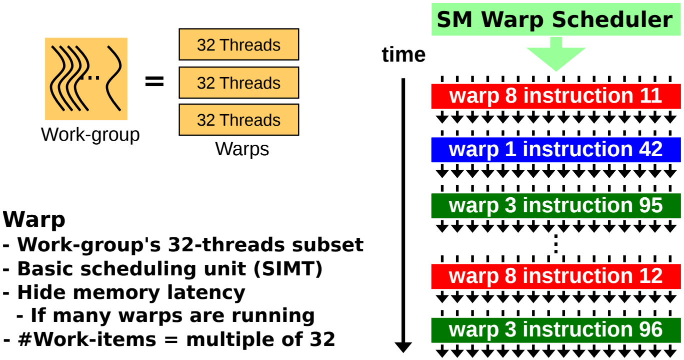
Work-items are executed by Scalar Processors

Work-groups are executed on Multiprocessors

A kernel is executed on Device as a Grid of work-groups

Figure : Software vs. Hardware layers.

Warps



Warp
- Work-group's 32-threads subset
- Basic scheduling unit (SIMT)
- Hide memory latency
- If many warps are running
- #Work-items = multiple of 32

Figure : Warps.

Global Memory Access

Global Memory is a set of aligned segments.

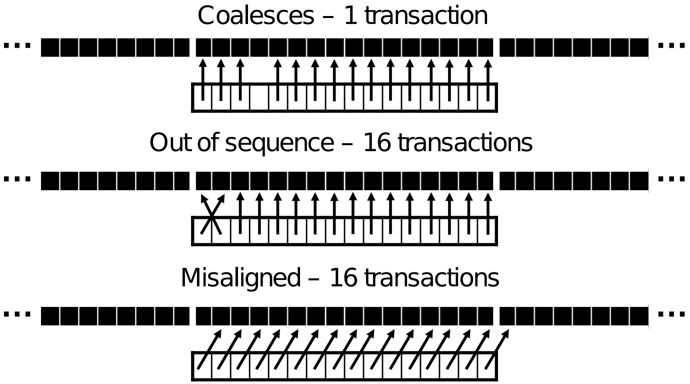

Coalesces – 1 transaction
Out of sequence – 16 transactions
Misaligned – 16 transactions

Figure : Memory access for CC 1.0/1.1.

Global Memory Access

Global Memory is a set of aligned segments.

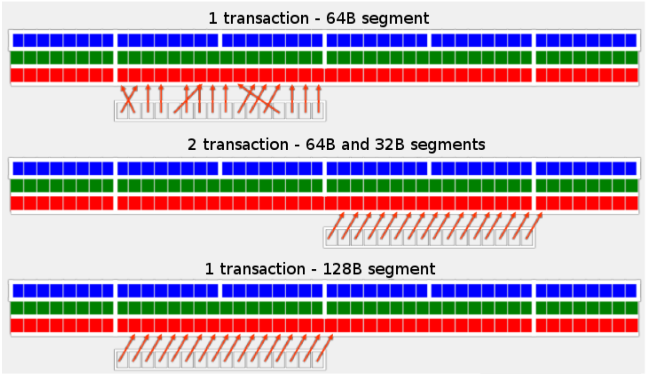

1 transaction - 64B segment
2 transaction - 64B and 32B segments
1 transaction - 128B segment

Figure : Memory access for CC 1.2/1.3.

Local Memory Access

- Local Memory is divided into banks.
- Work-items can access local memory simultaneously.
- Access to same memory bank cause serialization.
    - Except when doing Broadcast.
- Some bank conflicts can be avoided by adding strides.

Local Memory Access
Bank Conflict

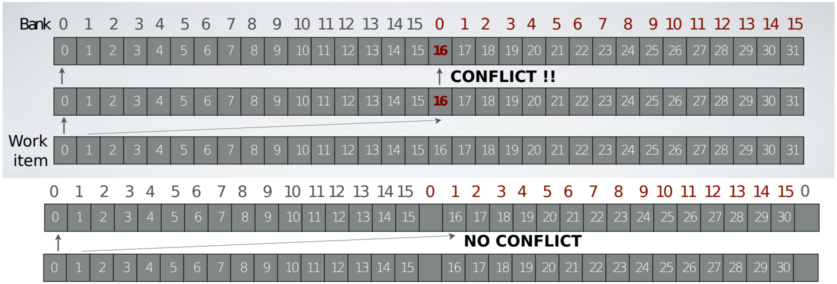


Figure : Stride on Matrix Transpose.

Local Memory Access
Bank Conflict

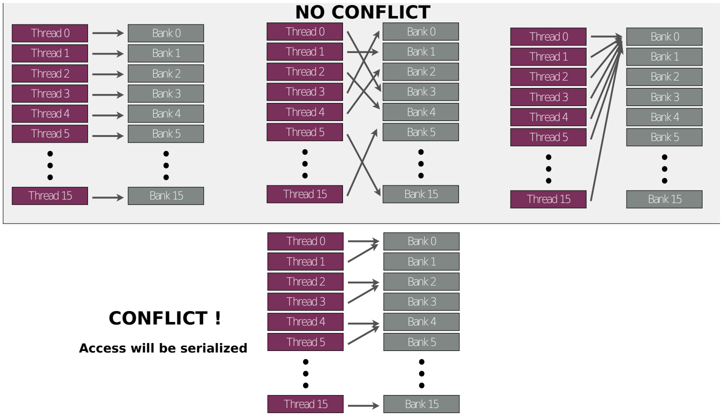


Figure : Bank Conflict on Local memory access.

Local Memory Access

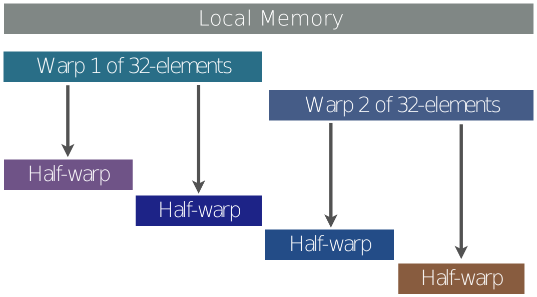


Figure : Local memory for a work-group of 2-warps size.

- No bank conflicts for half-warp (16 banks).
- Barriers are important to keep the synchronism.

Memory Access

- Global memory is visible to all work-items on device.
  - 400 to 600 cycles of memory latency.
- Local (shared) memory is visible among all work-items within the same work-group.
  - Latency about 100x smaller than global memory.
- Each Work-item (Thread) has its own Private memory.
  - Latency zero.

Memory Access

- Accessing Private memory (registers) has zero extra clock cycle per instruction.
    - Register pressure and long data structures on private area cause variables to be moved automatically by compiler to a special Global memory section.
    - Read-after-write on registers has a dependency of about 24 cycles and therefore must be avoided, but multiple warps (> 6) can hide totally this latency.

Work-item Grouping

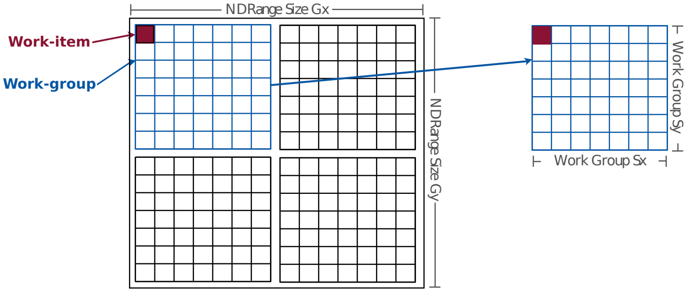


Figure : Work-items and Work-groups.

Work-item Grouping

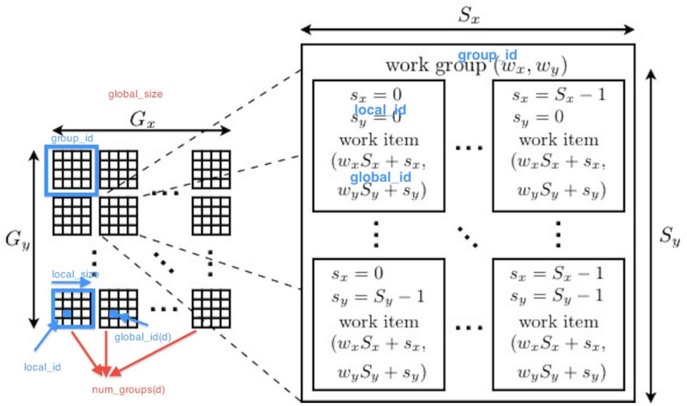


Figure : Work-item mapping functions.

- **clWaitForEvents(num_events, event_list):** Host blocks until all events in wait list are finished.
- **clFinish:** Host waits for all enqueued event to finish.
- **clEnqueueWaitForEvents(queue, num_events, event_list):** Command queue waits for all events in wait list to finish. Events do not need to be from same queue.
- **mem_fence(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE):** Force work-item completion of loads and stores before any new load/store can be executed. The functions **read_mem_fence()** and **write_mem_fence()** are the equivalent for only load and store, respectively.
- **barrier(mem_fence_flag):** Work-item waits untill all work-items in work-group reach the barrier. It also triggers a memory fence.

- **__local:** Creates a local (shared) variable, whose contents is visible to all work-items in same work-group.
  - However, it must be statically allocated within the kernel (numeric literal, constant, macro, etc).
- To allocate local memory dynamically, set it as a parameter when enqueueing the kernel.

On cl file:

```
__kernel void theKernel(
    __local float *dynamicAllocation){...}
```

On kernel call:

```
clSetKernelArg(kernel, 0,
    local_dynamic_size_in_bytes, NULL);
```

- Commands are submitted to OpenCL through command queues.
- A Queue is tied to a device.
- Multiple queues can be active concurrently.
  - Only one command running per device at a time.
- Commands are executed at the order they were submitted.
  - CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE: Queues created with this property will try to execute the commands at a more optimized sequence.

OpenCL Framework

Download and compile the framework from Git repository.

```
$ wget --no-check-certificate https://gitlab.c3sl.ufpr.br/
pfperroni/oclframework/repository/archive.tar.gz?ref=master -O
oclFramework.tar.gz
$ tar -xzvf oclFramework.tar.gz
$ cd oclframework.git/src
$ ./make_all.sh
```

Test a GPU call.

```
$ cd Examples/SimpleGPUCall
$ ./SimpleGPUCall
```

Matrix Multiplication

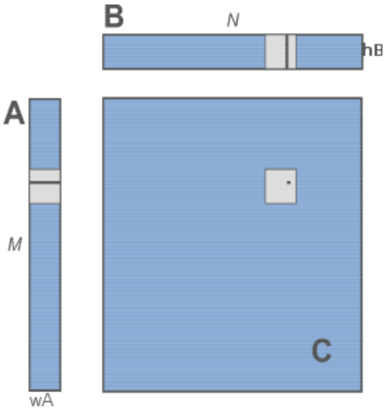


Figure : Simple Matrix multiplication.

Matrix Multiplication
Host code

```
float *A = new float[m*wA];
float *B = new float[hB*n];
float *C = new float[m*n]

for(int c, r=0; r < m; r++){
    for(c=0; c < n; c++){
        for(sum=0, i=0; i < wA; i++){
            sum += A[r * wA + i] * B[i * n + c];
        }
        C[r * n + c] = sum;
    }
}
```

Matrix Multiplication
Host code

Implement the matrix multiplication and measure the run time with function below.
- Use dimensions 1024x1024 for A, B and C.
- Initialize the matrices with value 1.

```
double timestamp(){
   struct timeval tp;
   gettimeofday(&tp, NULL);
   return (double)(tp.tv_sec +
      tp.tv_usec / 1000000.0);
}
```

Matrix Multiplication
OpenCL code

Startup OpenCL environment.

```
clFactory::setDeviceType(RUN_ON_GPU);
clFactory::startup();
```

Compile cl code.

```
vector<char*> kernelNames;
kernelNames.push_back((char*)"simpleMatrixMul");
startupKernels((char*)"matrix_mult.cl", kernelNames);
```

Get a reference to the compiled kernel.

```
cl_command_queue command_queue = queue->getCommandQueue();
cl_device_id device = queue->getDevice();
kernel_t* kernel =
    getKernelInstanceByDevice((char*)"simpleMatrixMul", device);
```

Matrix Multiplication
OpenCL code

Create host variables.

```
int wA = 1024, m = 1024, hB = 1024, n=1024;
WORD *A = new WORD[m*wA];
WORD *B = new WORD[hB*n];
WORD *C = new WORD[m*n];
```

Create device variables.

```
cl_context context = queue->getContext();
CREATE_BUFFER(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
    (m*wA) * sizeof(float), A, cl_A);
CREATE_BUFFER(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
    (hB*n) * sizeof(float), B, cl_B);
CREATE_BUFFER(context, CL_MEM_READ_WRITE,
    (m*n) * sizeof(float), NULL, cl_C);
```

Matrix Multiplication
OpenCL code

Enqueue the kernel.

```
// M x N Threads, 1 per each element of matrix C.
CALL_KERNEL2D(command_queue, kernel, m, n, blkSz, blkSz, 5,
    sizeof(cl_mem), (void*)&cl_A,
    sizeof(cl_mem), (void*)&cl_B,
    sizeof(cl_mem), (void*)&cl_C,
    sizeof(cl_int), (void*)&wA,
    sizeof(cl_int), (void*)&n
);
```

Wait for the kernel to finish.

```
SYNC_QUEUE(command_queue);
```

Matrix Multiplication
OpenCL code

Copy the results back to host.

```
clMemcpyDeviceToHost(command_queue, C, cl_C,
    (m*n) * sizeof(float));
```

Release the device and host variables.

```
clReleaseMemObject(cl_A);
clReleaseMemObject(cl_B);
clReleaseMemObject(cl_C);
delete A;
delete B;
delete C;
```

Shutdown OpenCL.

```
clFactory::shutdown();
```

Matrix Multiplication
OpenCL code

Receiving parameters.

```
__kernel void simpleMatrixMul(
    __global WORD* A, // matrix A
    __global WORD* B, // Matrix B
    __global WORD* C, // Resulting matrix C
    int wA, int N // A and B width, respectively.
)
```

Obtain work-item positioning on global thread mapping.

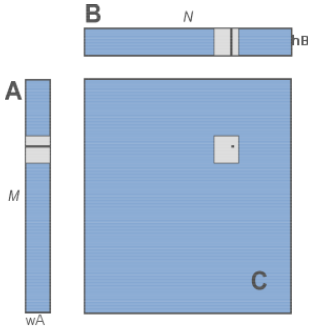
```
// Global index
int gx = get_global_id(0); // C matrix column.
int gy = get_global_id(1); // C matrix row.
```

Matrix Multiplication
OpenCL code

```
// Create a private variable (register)
// to accumulate the sum.
float sum = 0;

 // Sum matrices A and B.
 for(int i=0; i < N; i++){
    sum += A[gy * wA + i] *
           B[i * N + gx];
 }

 // Send the sum back
 // to Global memory.
 C[gy * N + gx] = sum;
```



Matrix Multiplication
OpenCL code

Using SimpleGPUCall as template, implement a matrix multiplication.

Matrix Multiplication
OpenCL code

Using SimpleGPUCall as template, implement a matrix multiplication.

Set the code to run on CPUs (instead of GPU), then compare the performance.

Matrix Multiplication
OpenCL code

Using SimpleGPUCall as template, implement a matrix multiplication.

Set the code to run on CPUs (instead of GPU), then compare the performance.

Now, re-run the versions with larger matrices sizes (2048x2048) and compare GPU vs CPU versions.

Optimized Matrix Multiplication

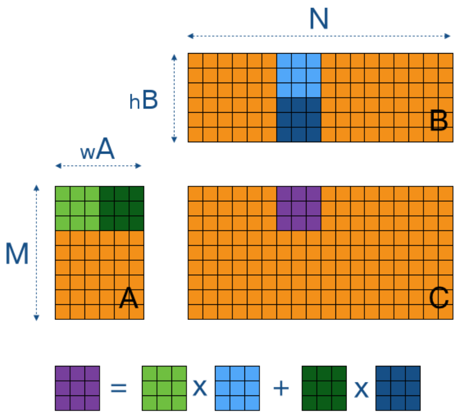


Figure : Matrix block multiplication.

Optimized Matrix Multiplication
OpenCL code

```
__kernel void optimizedMatrixMul(__global WORD* A,
    __global WORD* B, __global WORD* C,
    int wA, int N, int blkSize,
    // Declaration of local memory arrays, used to store the
    // sub-matrices of A and B.
    __local WORD *As, __local WORD *Bs){

   // Block index
   int bx = get_group_id(0);
   int by = get_group_id(1);
   // Thread index
   int tx = get_local_id(0);
   int ty = get_local_id(1);
```

Optimized Matrix Multiplication
OpenCL code

```
// Index of the first sub-matrix of A to be processed
// by the block
int aBegin = wA * blkSize * by;

// Index of the last sub-matrix of A to be processed
// by the block
int aEnd = aBegin + wA - 1;

// Step size used to iterate through the
// sub-matrices of A
int aStep = blkSize;
```

Optimized Matrix Multiplication
OpenCL code

```
// Index of the first sub-matrix of B to be processed
// by the block
int bBegin = blkSize * bx;

// Step size used to iterate through the
// sub-matrices of B
int bStep = blkSize * N;

// Register to accumulate the sum.
float sum = 0;
```

Optimized Matrix Multiplication
OpenCL code

```
// Main loop.

for(int a=aBegin, b=bBegin; a <= aEnd; a+=aStep, b+=bStep){
   // Bring block to local memory.
   As[ty * blkSize + tx] = A[a + wA * ty + tx];
   Bs[ty * blkSize + tx] = B[b + N * ty + tx];

   barrier(CLK_LOCAL_MEM_FENCE);

   for(int k=0; k < blkSize; ++k){
      sum += As[ty * blkSize + k] * Bs[k * blkSize + tx];
   }

   barrier(CLK_LOCAL_MEM_FENCE);
}

// Write the block sub-matrix to device memory.
// Each thread writes exactly one element.
C[get_global_id(1) * N + get_global_id(0)] = sum;
```

Optimized Matrix Multiplication

Implement the optimized matrix multiplication and run it with matrix sizes of:
- 1024x1024
- 2048x2048

Optimized Matrix Multiplication

Implement the optimized matrix multiplication and run it with matrix sizes of:
- 1024x1024
- 2048x2048

Set the code to run on CPUs (instead of GPU), then compare the performance with all other versions.