OpenCL

Politecnico di Milano
Seminar Room A. Alario
20 November, 2015

Antonio R. Miele
Marco D. Santambrogio
Politecnico di Milano
Modern architectures are heterogeneous

- A modern computer architecture includes
  - One or more CPUs
  - One or more GPUs
  - Optionally other HW accelerators

- Architectures may have various computing unit per type
- This aspect enables parallelism!
Programmability issues

• Each type of resource is generally programmed by using different languages and paradigms

• CPU
  – A huge variety of languages is available: C, C++, Java, ...
  – Various APIs for expressing parallelism: OpenMP, pthreads, ...

• GPU
  – CUDA is a C-based language for NVIDIA GPUs
  – C++ AMP is a programming model based on DirectX defined by Microsoft

• FPGA
  – Verilog and VHDL are the two commonly used HDL to define accelerators
  – Some vendors define high-level proprietary languages for their boards (e.g. MaxJ defined by Maxeler)

• OpenMP and OpenACC are two directive-based languages for parallel computing that recently started supporting heterogeneous devices (CPU, GPU, further accelerators…). Not very flexible in the management of the device

• Huge heterogeneity in system programming!
OpenCL

- OpenCL (Open Computing Language) is a programming framework for heterogeneous compute resources
- Original issue has been to solve heterogeneity between CPUs and GPUs
OpenCL

- OpenCL is a programming framework for heterogeneous compute resources
- Nowadays OpenCL targets many kind of devices
OpenCL implementations
OpenCL working group

- Diverse industry participation – many industry experts
  - Processor vendors, system OEMs, middleware vendors, application developers
- Apple made initial proposal and is very active in the working group
  - Serving as specification editor
OpenCL timeline

- OpenCL initially developed by Apple
- Meanwhile managed by Khronos Group, a non-profit technology consortium

- We will present OpenCL 1.2 and provide some details on the novelties introduced by OpenCL 2.0 and 2.1
Design goals of OpenCL

• Low-level abstraction for compute
  – Avoid specifics of the hardware (but not hiding hardware management)
  – Enable high performance
  – Support device independence

• Enable all compute resources
  – CPUs, GPUs, and other processors
  – Data- and task-parallel compute models
  – Efficient resource sharing between processors
Design goals of OpenCL

• Efficient parallel programming model
  – ANSI C99 based kernel language
  – Minimal restrictions and language additions
  – Straight-forward development path

• Consistent results on all platforms
  – Well-defined precision requirements for all operations
  – Maximal-error allowances for floating-point math
  – Comprehensive conformance suite for API + language
OpenCL platform model

- The OpenCL platform model is a high-level description of the heterogeneous systems
  - Abstract from peculiarities of specific vendors to have a unified model
OpenCL platform model

- One host connected to a set of compute devices
  - Each compute device is composed of one or more compute units
  - Each compute unit is further divided into one or more processing elements
  - The single unit of computation is executed by a processing element
OpenCL platform model

- One host connected to a set of compute devices
  - Each compute device is composed of one or more compute units
  - Each compute unit is further divided into one or more processing elements
  - The single unit of computation is executed by a processing element

Mapping on NVIDIA GPU

In ARM chips, the GPU may be partitioned in various devices
OpenCL application

- **Device code**
  - Written in OpenCL C
  - Organized in kernels
  - Executes on the device
  - Data-parallel execution

- **Host code**
  - Written in C/C++
  - Executes on the host
  - Mainly sequential execution
  - May use task-parallel execution

Host code sends commands to the devices:
- To transfer data between host memory and device memories (kernel codes and data)
- To execute kernels on the device
OpenCL application

- Serial code executes in a host thread on the host (i.e. a CPU)
- Parallel code (that is the kernel) executes in many device threads across multiple processing elements on the device (e.g. a GPU)
Decompose a task into parallel work-items

- The kernel is actually a data-parallel C function executed on a single instance of the problem

- Parallelization approach:
  - Define an N-dimensional integer index space (N=1,2,3) called NDRange
  - Execute a kernel at each point in the integer index space

```c
void
trad_mul(int n,
           const float *a,
           const float *b,
           float *c)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

```c
_kernel void
dp_mul(__global const float *a,
       __global const float *b,
       __global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over n “work items”
```
Decompose a task into parallel work-items

- The kernel is actually a data-parallel C function executed on a single instance of the problem
- Parallelization approach:
  - Define an N-dimensional integer index space (N=1,2,3) called NDrange
  - Execute a kernel at each point in the integer index space

Traditional loop-based C function

```c
void trad_mul
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] * b[i];
}
```

OpenCL parallel function

```c
_kernel void
dp_mul(__global const float *a,
      __global const float *b,
      __global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over n "work items"
```
Decompose a task into parallel work-items

- **Work-item**: single execution of the kernel on a data instance (i.e. the basic unit of work)
- **Work-group**: group of contiguous work-items
- Work-groups have the same size and exactly span the NDrange
- The NDrange is actually an N-dimensional grid with a local and global indexi
Decompose a task into parallel work-items

- Example of 1 dimensional problem

```c
kernel void square(global float* input, global float* output)
{
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

- `get_work_dim = 1`
- `get_global_size = 16`
- `get_num_groups = 2`
- `get_group_id = 0`
- `get_local_size = 8`
- `get_local_id = 3`
- `get_global_id = 11`
Decompose a task into parallel work-items

- Example of 3 dimensional problem
Kernel execution on platform model

- Each work-item is executed by a compute element
- Each work-group is executed on a single compute unit
- A wavefront or warp (depending on the implementation) is the subgroup of work-items of a single work-group executed concurrently in lockstep
- Each kernel is executed on a compute device

- Several concurrent work-groups can reside on one compute unit depending on work-group’s memory requirements and compute unit’s memory resources
Supported parallelism models

- **Single-Instruction-Multiple-Data (SIMD)**
  - The kernel is composed of sequential instructions
  - The instruction stream is executed in lock-step on all involved processing elements
  - Generally used on GPU

- **Single-Program-Multiple-Data (SPMD)**
  - The kernel contains loops and conditional instruction
  - Each processing element has its own program counter
  - Each work-item has a different execution flow
  - Generally used on CPU
Supported parallelism models

• Data-parallel programming model
  – The same sequential instruction stream is executed in lock-step by many work-items
  – Generally used on GPU

• Task-parallel programming model
  – There is one work-item per core (or few ones) per kernel
  – The program issues many kernels in a out-of-order fashion
  – Generally used on CPU
OpenCL memory model

- Shared memory model
  - With relaxed consistency
  - Explicit synchronization is required

- Private memory
  - Per work-item

- Local memory
  - Shared within a work-group for sharing data

- Global/constant memory
  - Visible to all work-items in any work-groups
  - Constant memory is initialized by the host and is read-only for the kernel

- Host memory
  - On the CPU

- Implementation maps this hierarchy to the available physical memories
OpenCL memory model

- Memory management is EXPLICIT
- Programmer has to move data from host -> global -> local … and back
  - With a memory copy, or
  - With a map of an OpenCL memory objects into the host memory address space
- Programmer has to take care of memory consistency between the kernel and the host and among work-items
OpenCL memory model

- OpenCL uses a “relaxed consistency memory model”:
  - State of memory visible to a work-item **not** guaranteed to be consistent across the collection of work-items at all times

- Memory has load/store consistency within a work-item
- Local memory has consistency across work-items within a work-group at an explicit synchronization point
- Global memory is consistent within a work-group at a an explicit synchronization point, but not guaranteed across different work-groups
- Global memory consistency between host and kernel and between different kernels must be enforced at explicit synchronization points

- The relaxed memory consistency model allows devices scalability
OpenCL framework

- Context
  - Programs
  - Kernels
  - Memory Objects
  - Command Queues

- CPU
  - GPU

- Compile code
- Create data & arguments
- Send to execution

__kernel void__

dp_mul(global const float *a, 
global const float *b, 
global const float *c) 
{ 
    int id = get_global_id(0); 
    c[id] = a[id] * b[id]; 
}
OpenCL framework

- A **device** represents an OpenCL compute device
OpenCL framework

- **A program object** contains the source code that implements the kernel and its compiled binary code.

---

Compile code  
Create data & arguments  
Send to execution
OpenCL framework

A **kernel** represents an instance of the execution of a program object.
It contains also the actual parameters.
OpenCL framework

- A **memory object** is a buffer (or a more advanced structure) allocated in the host or device memory to exchange data between the application and the kernel.
OpenCL framework

- The **command queue** is used by the host application to submit work to a single device (e.g., kernel execution instances)
- Work is queued in-order, one queue per device
- Work is executed asynchronously w.r.t. the host application
- Work can be executed in-order or out-of-order
OpenCL framework

- The **context** is the environment within a kernel is defined and executed.
- It includes devices, kernels, memory objects and command queues.
Execution flow

- The application discovers and setups the execution context
  - Discover compute devices
  - Compile kernels
Execution flow

• The application copies the data from the host memory to the device memory by using memory objects
Execution flow

• The application sets the kernel parameters and launches the execution of the kernel on the GPU
Execution flow

• The GPU executes the kernel on the data stored in its device memory
Execution flow

- The application copies back the data from the device memory to its memory by using the memory objects.
OpenCL stack

- Many vendors provide OpenCL implementations
  - There are also open-source implementations
- The OpenCL stack has
  - The front-end is equal for all implementations
  - The back-end is vendor-specific
OpenCL stack

- In order to support multiple OpenCL implementations
  - Each implementation provides an Installable Client Driver (ICD)
  - The application is dynamically linked to the ICD loader that manages available ICDs and forward requests
OpenCL Compilation System

- OpenCL uses a Dynamic/Runtime compilation model for the kernel code
  - The code is compiled to an Intermediate Representation (IR)
  - The IR is compiled to a machine code for execution
- LLVM is used for the front-end (replaced/integrated with SPIR-V in OpenCL 2.1)
- Vendor-specific back-ends are used for the second phase
- The host code is still compiled statically
OpenCL language and API

• Let’s see the source code...

... of the OpenCL kernel
Definition of an OpenCL kernel

• The kernel is implemented in OpenCL language, an extension of C99

• Restrictions w.r.t. C99
  – NO Function pointers
  – NO recursion
  – NO variable-length arrays
  – … few others
Definition of an OpenCL kernel

• New data type
  – Scala data types
    • half, ...
    • Implicit/explicit casts
  – Image types
    • image2d_t, image3d_t, sampler_t
  – Vector data types
    • char2, ushort4, int8, float16, double2, ...
    • Vector lengths 2, 3, 4, 8, 16
    • Endian safe
    • Aligned at vector length
    • Vector operations (+, -, *, -, access to fields, implicit/explicit casts...)
Definition of an OpenCL kernel

• Address space qualifiers
  – __global, __local, __constant, __private

• Built-in functions
  – Work-items functions: get_global_id, get_local_id, ...
  – Math functions: sin, cos, exp, ...
  – Common functions: min, clamp, max, ...
  – Geometric functions: cross, dot, normalize, ...
  – Vector load/store functions: vload4, vstore4, ...
  – Synchronization functions: barrier
  – Atomic functions: atomic_add, atomic_sub, ...
Definition of an OpenCL kernel

• Example:

```c
__kernel void dp_mult(__global const float* a,
                      __global const float* b,
                      __global float* c)
{
    int i = get_global_id(0);
    c[i] = a[i] * b[i];
}
```
Definition of an OpenCL kernel

• Address space qualifiers
  – __global, __local, __constant, __private

```c
__constant float myarray[4]={...};
__kernel void pDFA {
  __constant int * InputLength,
  __local   int * Matched,
  __global  int * InputStr,
  int OtherThanThat
} {
  int k, j;
  // Declares a pointer variable p in the
  // private address space. The pointer
  // points to an int object in address
  // space __global:
  __global int *p;
}
```

- Variables outside of the kernel scope must be constant.
- Kernel arguments that are pointers can point to constant, local or global memory.
- All non-pointer kernel arguments are private.
- All variables declared inside of a kernel function are private.
- But pointers declared inside of a kernel function can point to other memory spaces!
Work-item synchronization

- Work-items within the same work-group can be synchronized by using a barrier
  - Necessary when work-items cooperate
- Work-items belonging different work-groups CANNOT be synchronized
  - To achieve global synchronization it is necessary to split the kernel in two subsequent ones
- `barrier()` sets a barrier within the code of a kernel. When all work-items reaches the barrier, the memory (local or global) is flushed and execution is resumed
OpenCL language and API

• Let’s see the source code...

... of the host application
Get platform

- A platform is a vendor-specific implementation of the OpenCL API
  - It contains a set of devices

- `clGetPlatformIDs()` is used for accessing the available platforms
- `clGetPlatformInfo()` is used for querying the information about the available platforms
Get platform

• Example:

```c
// get all platforms
cl_int err;
cl_uint numPlatforms;
cl_platform_id *platformIds;
err = clGetPlatformIDs(0, NULL, &numPlatforms);
platformIds = (cl_platform_id *)
    malloc(sizeof(cl_platform_id) * numPlatforms);
err = clGetPlatformIDs(numPlatforms, platformIds, NULL);
```

• All `clGet*` functions can be used to access either the number of items or the list of items
  – Specific parameter lists distinguish the two queries
Get platform

• Example:

```c
//get the name of a platform given its id
err = clGetPlatformInfo(id, CL_PLATFORM_NAME, 0, NULL, &size);
char * name = (char *) malloc(sizeof(char) * size);
err = clGetPlatformInfo(id, CL_PLATFORM_NAME, size, name, NULL);
std::cout << "Platform name: " << name << std::endl
```

• Various kinds of information may be queried:

  - `CL_PLATFORM_NAME`, `CL_PLATFORM_VENDOR`, `CL_PLATFORM_PROFILE`, ...
Get device

• Each platform may contain one or more compute device

• `clGetDeviceIDs()` can be used for querying the available devices

• `clGetDeviceInfo()` can be used for querying the information about the devices
Get device

• Example:

```c
//get all devices of a given platform
cl_uint numDevices;
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 0,
                     NULL, &numDevices);
cl_device_id* deviceIds = (cl_device_id *)
                           malloc(sizeof(cl_device_id) * numDevices);
err = clGetDeviceIDs(platform, CL DEVICE TYPE_GPU,
                      numDevices, deviceIds, NULL);
```

• It is possible to filter on the device type
  - CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_CPU,
    CL DEVICE_TYPE ACCELERATOR
Get device

• Example:

```c
//get the maxComputeUnits of a specified device
cl_uint maxComputeUnits;
err = clGetDeviceInfo(deviceID, CL_DEVICE_MAX_COMPUTE_UNITS,
                      sizeof(cl_uint), &maxComputeUnits,
                      &size);
std::cout << "Device has max compute units: "
          << maxComputeUnits << std::endl;
```

• Various kinds of information may be queried:
  
  – CL_DEVICE_MAX_WORK_ITEM_SIZES,
    CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
    CL_DEVICE_MAX_WORK_GROUP_SIZE
Device partitioning

- Devices can be partitioned into sub-devices
  - More control over how computation is assigned to compute units
  - Sub-devices may be used just like a normal device
    - Create contexts, build programs, create command queues
  - Three ways to partition a device
    - Split into equal-size groups
    - Provide list of group sizes
    - Group devices sharing a part of a cache hierarchy
Create context

- A context provides a container for associated devices, program objects, kernels, memory objects and command queues
- We may instantiate multiple context within a program
- A context may contain various devices of the same platform

- Multiple contexts enable dynamic mapping of the kernels to be executed
  - Data transfers among different contexts have to be explicitly managed by the programmer
Create context

• A context may be created in two different ways
  - `clCreateContext()` creates a context from a specified list of devices of a single platform
  - `clCreateContextFromType()` creates a context by using a specified type of platform and device
Create context

• Example 1:

//select the first device from the specified platform
cl_context context;
cl_context_properties properties [] =
    {CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0};
context = clCreateContext(properties, 1, devices[0],
                          NULL, NULL, NULL);

• Example 2:

//select all devices of any type from the specified platform
cl_context context;
cl_context_properties properties[] =
    {CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0};
context = clCreateContextFromType(properties,
                                   CL DEVICE TYPE ALL, NULL, NULL, NULL);}
Create command queue

• A command queue is used for issuing commands to a device
• One command queue per device needed
  – Dispatching among devices within the same context explicitly managed by the programmer
  – A single device may have multiple independent queues
• Available commands
  – Copy data to/from device
  – Execute a kernel
  – Synchronize
• Within a single command queue:
  – Commands can be synchronous or asynchronous
  – Commands can be executed in-order or out-of-order
Create command queue

• `clCreateCommandQueue()` creates a command queue on a specified device within a context

• Example:

```c
// create a command queue on the first device within the related context
cl_command_queue queue;
queue = clCreateCommandQueue(context, devices[0], 0, &err);
```

• Commands issued in the queue are executed in order
  – The execution of a command starts when the previous one is completed
  – `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` enables out of order execution of the commands within the queue
Build program

- The source code of each kernel is a text string
  - It can be stored in a C/C++ string or in a text file
- The source code has to be loaded and compiled within the context it needs to be used

- `clCreateProgramWithSource()` creates a program object from the source code
- `clBuildProgram()` compiles the program to obtain the executable code on each device type within the selected context
Build program

• Example:

```c
//source code is created in the specified context and it
is compiler for the first device in the list
const char *program_source =
    "__kernel void dp_mult (__global const float *a,"
    " __global const float *b, __global float *c)\n",
    "...");

cl_program program;
program = clCreateProgramWithSourceWithEnvironment(context, 1,
    program_source, NULL, &err);
clBuildProgram(program, 1, devices, "", NULL, NULL)
```

• Further options:
  – Compiling for all devices within the context
  – Loading an already compiled binary (for a specified device)
Create a kernel

- In OpenCL language a kernel represents an instance of the execution of a program object
- `clCreateKernel()` creates a kernel from a program object

Example:
```c
//create a kernel
cl_kernel kernel = clCreateKernel(program, "dp_mult", &err);
```
Create memory objects

- Memory objects are used to transmit data to/from a device
  - The buffer is the basic 1-dimensional memory object
  - OpenCL provides also advanced 2/3-dimensional memory objects for managing images
- Memory objects are visible within a device but memory consistency has to be managed explicitly among different devices
- Memory objects cannot be shared among different contexts since they are potentially associated to different platforms
- Memory objects are accessed in the kernel through pointers
Create memory objects

- `clCreateBuffer()` creates a memory object
- Buffers may be read-only, write-only or read-write
- It is possible to initialize buffers with data

Example:

```c
//create buffers for input and output
cl_mem bufA, bufB, bufC;
bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, 
                        sizeof(float)*NUM_DATA, NULL, &err);
bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, 
                        sizeof(float)*NUM_DATA, NULL, &err);
bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, 
                        sizeof(float)*NUM_DATA, NULL, &err);
```
Write into memory objects

- `clEnqueueWriteBuffer()` transfers data from the host memory to the device global memory.

**Example:**

```c
// transfer input data
float a[NUM_DATA], b[NUM_DATA];
// write something in a and b
clEnqueueWriteBuffer(queue, bufA, CL_TRUE, 0,
                      sizeof(float)*NUM_DATA, a, 0, NULL, NULL);
clEnqueueWriteBuffer(queue, bufB, CL_TRUE, 0,
                      sizeof(float)*NUM_DATA, b, 0, NULL, NULL);
```

- An alternative is to map the memory object into the host address space so that the host application can access it.
Pass parameters

- Parameters have to be specified one by one
- `clSetKernelArg()` is used to specify a single parameter per time

Example:

```c
//specify the three parameters for the dp_mult function
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);
```
Execute the kernel

- `clEnqueueNDRangeKernel()` launches the execution of a kernel on a specified NDRange.

- Two possibilities for the work-group size:
  - Do not specify anything and OpenCL will manage it.
  - Explicitly specify it.
    - Best performance when the maximum work-group size supported by the device is used.
    - The considered problem may lead to a different value.

- Example:

```c
//executes the dp_mult kernel on the specified NDRange
size_t global_work_size[1] = { NUM_DATA };
size_t local_work_size[1] = { 32 };
clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
                      global_work_size, local_work_size, 0, NULL, NULL);
```
Read results

• `clEnqueueReadBuffer()` transfers data from the device global memory to the host memory

• Example:

```c
//read results
float results[NUM_DATA];
clEnqueueReadBuffer(queue, bufC, CL_TRUE, 0,
    sizeof(float)*NUM_DATA, results, 0, NULL, NULL));
```
Release resources

• At the end of the program, all OpenCL resources have to be released

• Example:

  //release resources
  clReleaseMemObject(bufA);
  clReleaseMemObject(bufB);
  clReleaseMemObject(bufC);
  clReleaseKernel(kernel);
  clReleaseProgram(program);
  clReleaseContext(context);
Further notes on the code

• Every OpenCL instruction returns an error code
  – Error codes have to be checked after every instruction to assess the correct execution
  – Otherwise the application will continue under an error condition

• Most of the source code is almost the same in every OpenCL application
More on work-items and work-groups sizing

- Do note: the work-groups must span exactly the global index space on every dimensions
- How to manage a problem with a size that is not divisible by a predefined work-group size?
- E.g. $\text{DATA\_SIZE}=109$ and $\text{workgroupsize}=16$
More on work-items and work-groups sizing

- A parameter specifying the problem size is added to the kernel

```c
__kernel void dp_mult(__global float* a,
                      __global float* b, __global float* c,
                      int N)
{
    int i = get_global_id(0);
    if(i<N)
        c[i] = a[i] * b[i];
}
```
More on work-items and work-groups sizing

• The global work size is round up to the first multiple of the work-group size greater than the problem size

    size_t local_work_size[1] = { 16 };
    size_t global_work_size[1] =
        { NUM_DATA +16 - NUM_DATA % 16 };

• Therefore the kernels with id>=NUM_DATA will not compute any multiplication
Branch divergence

- On a GPU, within a warp/wavefront, control-flow should not diverge among work items
  - Otherwise, a so-called branch divergence occurs
- GPU cannot follow two instruction streams at the same time within warp
  - Divergence between warps/wavefronts is ok!
To avoid branch divergence, conditions of if-statements must be re-worked in such a way that all work items within a warp/wavefront take the same branch.
Command synchronization

- Explicit synchronization management has to be performed to assess the desired precedencies in the kernel execution when using
  - an out-of-order command queue
  - different queues
Command synchronization

- **clEnqueueBarrier()** sets a barrier on the queue specified as parameter
  - Commands after the barrier start executing only after all commands before the barrier have completed

- **clFinish()** blocks the execution of the host program until all commands in the queue specified as parameters are completed

- Example:
  ```
  //... send commands
  clEnqueueBarrier(queue);
  //... send commands
  clFinish(queue);
  ```
Command synchronization

- **Events**
  - Commands return events and obey event waitlists
  - A `cl_eventobject` can be associated with each command (last three parameters)
    ```
    clEnqueue*(...,
               num_events_in_waitlist,
               *event_waitlist,
               *event);
    ```
    - Any commands (or `clWaitForEvents()`) can wait on events before executing
    - Event object can be queried to track execution status of associated command and get profiling information

- `clEnqueueReadBuffer`, `clEnqueueReadBuffer` can be optionally blocking
An example

- Let’s consider as example the multiplication of two matrixes $A(M,K)$ and $B(K,N)$ producing $C(M,N)$

$$C[i,j] = \text{sum} \ (A[m,k] \times B[k,n], \ k = 0, 1, \ldots, K)$$
An example

• Serial solution:

```c
void SerialMultiply (float A[P][Q], float B[Q][R],
                    float C[P][R])
{
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            for (int k = 0; k < Q; k++)
                C[i][j] += A[i][k] * C[k][j];
}
```

Just for sake of simplicity dimension management is performed by means of constant macros
An example

- Parallel solution:

```c
__kernel void parMultiply1 (  
  __global float *A, __global float *B,  
  __global float *C, int K)  
{
    // Vector element index
    const int m = get_global_id(0);
    const int n = get_global_id(1);
    const int M = get_global_size(0);
    C[i + M * j] = 0;
    for (int k = 0; k < K; k++)
        C[i+M*j] += A[i+M*k] * B[k+M*j];
}
```
An example

- Parallel solution using local memory:
  - Idea: split the matrix multiplication problem into submatrixes that fit in the local workgroup size
  - We can compute each 3x3 submatrix of the answer as the sum of products of the first submatrixes, as if the submatrixes were elements of a matrix
An example

- Parallel solution using local memory:

```c
// we assume to have work groups having TS x TS size
__kernel void parMultiply2 (__global float *A, __global float *B,
                           __global float *C, int K)
{
    const int row = get_local_id(0); // Local row ID (max: TS)
    const int col = get_local_id(1); // Local col ID (max: TS)
    const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
    const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)
    const int M = get_global_size(0);
    const int N = get_global_size(1);

    // Local memory to fit a tile of TS*TS elements of A and B
    __local float Asub[TS][TS];
    __local float Bsub[TS][TS];
```
An example

• Parallel solution using local memory: (continue…)

// Initialize the accumulation register
float acc = 0.0f;

// Loop over all tiles
const int numTiles = K/TS; // K must be a multiple of TS
for (int t=0; t<numTiles; t++) {
    // Load one tile of A and B into local memory
    const int tiledRow = TS*t + row;
    const int tiledCol = TS*t + col;
    Asub[col][row] = A[tiledCol*M + globalRow];
    Bsub[col][row] = B[globalCol*K + tiledRow];
    // Synchronize to make sure the tile is loaded
    barrier(CLK_LOCAL_MEM_FENCE);
An example

• Parallel solution using local memory: (continue…)

```c
// Perform the computation for a single tile
for (int k=0; k<TS; k++) {
    acc += Asub[k][row] * Bsub[col][k];
}

// Synchronize before loading the next tile
barrier(CLK_LOCAL_MEM_FENCE);
}

// Store the final result in C
C[globalCol*M + globalRow] = acc;
```
Conclusions

• OpenCL – Open Computing Language
  – Open, royalty-free standard
  – For cross-platform, parallel programming of modern processors
  – An Apple initiative
  – Approved by NVIDIA, Intel, AMD, IBM, ...
  – Specified by the Khronos group

• Intended for accessing heterogeneous computational resources
  – CPUs (Intel, AMD, IBM Cell BE, ...)
  – GPUs (Nvidia GTX & Tesla/Fermi, AMD/ATI 58xx, …)
  – HW accelerators
  – Processors with integrated graphics chip (Sandy Bridge, Llano)

• Difference to CUDA
  – Code hardware agnostic, portable
Conclusions

• Opportunities offered by OpenCL
  – Program with a single language an heterogeneous processing system
  – Decide at run-time where to execute each kernel according to the current working conditions
References

- https://www.khronos.org/
- https://developer.nvidia.com/opencl