An introduction to many-core parallel computing with OpenCL

Simon McIntosh-Smith

Twitter: @simonmcs

UPMARC summer school
Uppsala 28-29\textsuperscript{th} 2014
Recap

5 simple steps in a basic OpenCL program:

1. Define the **platform** = devices + context + queues
2. Create and Build the **program** (dynamic library of kernels)
3. Setup **memory** objects
4. Define the **kernels**
5. Submit **commands** ... transfer memory objects and execute kernels
We have now covered the basic platform runtime APIs in OpenCL

- **Context**
  - CPU
  - GPU

**Programs**
- __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];

**Kernels**
- `dp_mul`
- `dp_mul`
- `dp_mul`

**Memory Objects**
- Buffers
- Images

**Command Queues**
- In Order Queue
- Out of Order Queue

**Compile code**

**Create data & arguments**

**Send to execution**
OPENCL KERNEL PROGRAMMING
OpenCL C kernel language

- Derived from ISO C99
  - A few restrictions: no recursion, function pointers, functions in C99 standard headers (more later)
  - Preprocessing directives defined by C99 are supported (#include etc.)

- Built-in data types
  - Scalar and vector data types, pointers
  - Data-type conversion functions:
    - convert_type<_sat><_roundingmode>
  - Image types: image2d_t, image3d_t and sampler_t
OpenCL C Language Highlights

Function qualifiers

• **__kernel** qualifier declares a function as a kernel
  – I.e. makes it visible to host code
• Kernels can call other OpenCL functions
  – Not all OpenCL functions have to be marked as __kernels - they just won't be visible to the host

Address space qualifiers

• **__global, __local, __constant, __private**
• Pointer kernel arguments **must** be declared with an address space qualifier
• **__private** is default for variables declared inside a kernel
OpenCL C Language Highlights

Work-item functions ("n" indicates dimension - 0, 1, 2)

- **get_global_size(n)**: number of work-items
- **get_local_size(n)**: number of work-items in work-group
- **get_global_id(n)**: global work-item ID
- **get_local_id(n)**: work-item ID inside work-group
- **get_work_dim()**: number of dimensions in use (1, 2 or 3)
- **get_group_id(n)**: ID of work-group

Synchronization functions

- **Barriers** - all work-items within a work-group must execute the barrier function before any work-item can continue
- **Memory fences** - provides ordering between memory operations
OpenCL C Language Restrictions

- Pointers to functions are **not** allowed
- Pointers to pointers allowed **within** a kernel, but not as an argument to a kernel invocation
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported (yet!)
- Double types are **optional** in OpenCL v1.2, but the key word is reserved
  
  *(note: most implementations support double)*
Matrix multiplication: sequential code

We calculate $C = AB$, all matrices square, of size "Order" on each side.

```c
void mat_mul(int Order, float *A, float *B, float *C) {
    int i, j, k;
    for (i = 0; i < Order; i++) {
        for (j = 0; j < Order; j++) {
            for (k = 0; k < Order; k++) {
                // $C(i,j) = \sum_k A(i,k) \times B(k,j)$
                C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
            }
        }
    }
}
```

Specialized to square matrices

Dot product of a row of $A$ and a column of $B$ for each element of $C$
Matrix multiplication performance

• Serial C code on CPU (single core).

<table>
<thead>
<tr>
<th>Case</th>
<th>MFLOPS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential C (not OpenCL)</td>
<td>887.2</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Device is Intel® Xeon® CPU, E5649 @ 2.53GHz using the gcc compiler.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.
Matrix multiplication: sequential code

```c
void mat_mul(int Order, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < Order; i++) {
        for (j = 0; j < Order; j++) {
            for (k = 0; k < Order; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
            }
        }
    }
}
```
Matrix multiplication: OpenCL kernel (1/3)

```
__kernel void mat_mul(
  const int Order,
  __global float *A, __global float *B, __global float *C)
{
  int i, j, k;
  for (i = 0; i < Order; i++) {
    for (j = 0; j < Order; j++) {
      // C(i, j) = sum(over k) A(i,k) * B(k,j)
      for (k = 0; k < Order; k++) {
        C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
      }
    }
  }
}
```

Mark as a kernel function and specify memory qualifiers
Matrix multiplication: OpenCL kernel (2/3)

```c
__kernel void mat_mul(
    const int Order,
    __global float *A, __global float *B, __global float *C)
{
    int i, j, k;

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

    for (k = 0; k < Order; k++) {
        // C(i, j) = sum(over k) A(i,k) * B(k,j)
        C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
    }
}
```

Remove outer loops and set work-item co-ordinates
Matrix multiplication: OpenCL kernel (3/3)

__kernel void mat_mul(
  __global float *A, __global float *B, __global float *C)
{
  int i, j, k;
  i = get_global_id(0);
  j = get_global_id(1);
  Order = get_global_id(0);

  for (k = 0; k < Order; k++) {
    // C(i, j) = sum(over k) A(i,k) * B(k,j)
    C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
  }
}

Get "Order" from total number of work-items (global size)
Matrix multiplication: OpenCL kernel tweaked

Rearrange a bit and use a local scalar for intermediate C element values (a common optimization in Matrix Multiplication functions)

```c
__kernel void mmul(    int k;
    __global float *A,    int i = get_global_id(0);
    __global float *B,    int j = get_global_id(1);
    __global float *C)    int Order = get_global_size(0);

    float tmp = 0.0f;
    for (k = 0; k < Order; k++)
        tmp += A[i*Order+k]*B[k*Order+j];

    C[i*Order+j] = tmp;
}
```
Matrix multiplication host program

```c
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{
    // declarations (not shown)
    sz = N * N;
    std::vector<float> h_A(sz);
    std::vector<float> h_B(sz);
    std::vector<float> h_C(sz);

    cl::Buffer d_A, d_B, d_C;

    // initialize matrices and setup
    // the problem (not shown)

    cl::Context context(DEVICE);
    cl::Program program(context, util::loadProgram("matmul1.cl", true));

    auto mmul = cl::make_kernel
        <cl::Buffer, cl::Buffer, cl::Buffer>
        (program, "mmul");

    d_A = cl::Buffer(context, begin(h_A), end(h_A), true);
    d_B = cl::Buffer(context, begin(h_B), end(h_B), true);
    d_C = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * sz);

    mmul(cl::EnqueueArgs(queue, cl::NDRange(N, N)), d_A, d_B, d_C);

    cl::copy(queue, d_C, begin(h_C), end(h_C));

    // Timing and check results (not shown)
}
```
Matrix multiplication performance

- Matrices are stored in global memory.

<table>
<thead>
<tr>
<th>Case</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential C (not OpenCL)</td>
<td>887.2</td>
</tr>
<tr>
<td>C(i,j) per work-item, all global</td>
<td>3,926.1</td>
</tr>
</tbody>
</table>

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Third party names are the property of their owners.
EXPLOITING THE OPENCL MEMORY HIERARCHY
OpenCL Memory model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a work-group
- **Global/Constant Memory**
  - Visible to all work-groups
- **Host memory**
  - On the CPU

Memory management is **explicit**:
You are responsible for moving data from host → global → local and back
Optimizing matrix multiplication

- Matrix multiply often benefits from reusing data as much as possible.

- So let’s have each work-item compute a sub block of $C$

$$C(ib,jb) = C(ib,jb) + A(ib,:) \times B(:,jb)$$

- We'll need to do something new:
  - Cache the sub blocks of A and B in local memory
  - Have to ensure local memory consistency (using barriers)
Memory Consistency

• OpenCL uses a relaxed consistency memory model; i.e.
  – The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.

• Within a work-item:
  – Memory has load/store consistency to the work-item’s own view of memory, i.e. it sees its own reads and writes correctly

• Within a work-group:
  – Local memory is consistent between work-items at a barrier.

• Global memory is consistent within a work-group at a barrier, but not guaranteed across different work-groups!!
  – This is a common source of bugs!

• Consistency of memory shared between commands (e.g. kernel invocations) is enforced by synchronization (barriers, events, in-order queue)
Work-Item Synchronization

• Within one work-group
  ```c
  void barrier()
  ```
  – Takes optional flags `CLK_LOCAL_MEM_FENCE` and/or `CLK_GLOBAL_MEM_FENCE`
  – A work-item that encounters a `barrier()` will wait until ALL work-items in its work-group reach the `barrier()`
  – **Corollary**: If a `barrier()` is inside a branch, then the branch **must** be taken by either:
    • **ALL** work-items in the work-group, OR
    • **NO** work-item in the work-group

• Between different work-groups
  – No guarantees as to where and when a particular work-group will be executed relative to another work-group
  – **Cannot exchange data, or have barrier-like synchronization between two different work-groups!** (Critical issue!)
  – **Only solution**: finish the kernel and start another

Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)
Blocked matrix multiply: kernel

```c
#define blksz 16
__kernel void mmul(
    const unsigned int N,
    __global float* A,
    __global float* B,
    __global float* C,
    __local  float* Awrk,
    __local  float* Bwrk)
{
    int kloc, Kblk;
    float Ctmp=0.0f;

    // compute element C(i,j)
    int i = get_global_id(0);
    int j = get_global_id(1);

    // Element C(i,j) is in block C(lblk,Jblk)
    int lblk = get_group_id(0);
    int Jblk = get_group_id(1);

    // C(i,j) is element C(iiloc, jiloc)
    // of block C(lblk, Jblk)
    int iloc = get_local_id(0);
    int jloc = get_local_id(1);
    int Num_BLK = N/blksz;

    // calc. upper-left-corner and inc. for A and B
    int Abase = lblk*N*blksz;
    int Ainc = blksz;
    int Bbase = Jblk*blksz;
    int Binc = blksz*N;

    // C(lblk,Jblk) = (sum over Kblk) A(lblk,Kblk)*B(Kblk,Jblk)
    for (Kblk = 0; Kblk<Num_BLK; Kblk++)
    {
        //Load A(lblk,Kblk) and B(Kblk,Jblk).
        //Each work-item loads a single element of the two
        //blocks which are shared with the entire work-group
        Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
        Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];
        barrier(CLK_LOCAL_MEM_FENCE);

        #pragma unroll
        for(kloc=0; kloc<blksz; kloc++)
            Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];
        barrier(CLK_LOCAL_MEM_FENCE);
        Abase += Ainc;
        Bbase += Binc;
    }
    C[j*N+i] = Ctmp;
}
```
```c
#define blksz 16
__kernel void mmul(
    const unsigned int N,
    __global float* A,
    __global float* B,
    __global float* C,
    __local  float*  Awrk,
    __local  float*  Bwrk)
{
    int kloc, Kblk;
    float Ctmp=0.0f;

    // compute element C(i,j)
    int i = get_global_id(0);
    int j = get_global_id(1);

    // Element C(i,j) is in block C(Iblk,Jblk)
    int Iblk = get_group_id(0);
    int Jblk = get_group_id(1);

    // C(i,j) is element C(i,loc, jloc)
    // of block C(Iblk, Jblk)
    int iloc = get_local_id(0);
    int jloc = get_local_id(1);
    int Num_BLK = N/blksz;

    // calc. upper-left-corner and inc. for A and B
    int Abase = Iblk*N*blksz;
    int Ainc = blksz;
    int Bbase = Jblk*blksz;
    int Binc = blksz*N;

    // C(Iblk,Jblk) = (sum over Kblk) A(Iblk,Kblk)*B(Kblk,Jblk)
    for (Kblk = 0; Kblk<Num_BLK; Kblk++)
    {
        //Load A(Iblk,Kblk) and B(Kblk,Jblk).
        //Each work-item loads a single element of the two
        //blocks which are shared with the entire work-group
        Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
        Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

        barrier(CLK_LOCAL_MEM_FENCE);

        #pragma unroll
        for(kloc=0; kloc<blksz; kloc++)
            Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

        barrier(CLK_LOCAL_MEM_FENCE);
        Abase += Ainc;  Bbase += Binc;
    }
    C[j*N+i] = Ctmp;
}
```

It's getting the indices right that makes this hard.
Blocked matrix multiply: Host

```c++
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{
    // declarations (not shown)
    sz = N * N;
    std::vector<float> h_A(sz);
    std::vector<float> h_B(sz);
    std::vector<float> h_C(sz);

    cl::Buffer d_A, d_B, d_C;

    // initialize matrices and setup
    // the problem (not shown)

    cl::Context context(DEVICE);
    cl::Program program(context,
                          util::loadProgram("mmulblock.cl", true));

    cl::CommandQueue queue(context);

    auto mmul = cl::make_kernel
                <int, cl::Buffer, cl::Buffer, cl::Buffer,
                 cl::LocalSpaceArg, cl::LocalSpaceArg>
                (program, "mmul");

    d_A = cl::Buffer(context, begin(h_A), end(h_A), true);
    d_B = cl::Buffer(context, begin(h_B), end(h_B), true);
    d_C = cl::Buffer(context,
                      CL_MEM_WRITE_ONLY, sizeof(float) * sz);

    cl::LocalSpaceArg Awrk =
                            cl::Local(sizeof(float) * N);
    cl::LocalSpaceArg Bwrk =
                            cl::Local(sizeof(float) * N);

    mmul(cl::EnqueueArgs(queue,
                          cl::NDRange(N,N), cl::NDRange(16,16)),
         N, d_A, d_B, d_C, Awrk, Bwrk);

    cl::copy(queue, d_C, begin(h_C), end(h_C));

    // Timing and check results (not shown)
}
```
```c
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{
    // declarations (not shown)
    sz = N * N;
    std::vector<float> h_A(sz);
    std::vector<float> h_B(sz);
    std::vector<float> h_C(sz);

    cl::Buffer d_A, d_B, d_C;

    // initialize matrices and setup
    // the problem (not shown)
    cl::Context context(DEVICE);
    cl::Program program(context, util::loadProgram("mmulblock.cl", true));

    auto mmul = cl::make_kernel
                 <int, cl::Buffer, cl::Buffer, cl::Buffer,
                  cl:::LocalStorageArg, cl:::LocalStorageArg >
                 (program, "mmul");

    d_A  = cl::Buffer(context, begin(h_A), end(h_A),true);
    d_B  = cl::Buffer(context, begin(h_B), end(h_B),true);
    d_C  = cl::Buffer(context,
                      CL_MEM_WRITE_ONLY, sizeof(float) * sz);

    cl::LocalStorageArg Awrk =
        cl::Local(sizeof(float) * 16*16);
    cl::LocalStorageArg Bwrk =
        cl::Local(sizeof(float) * 16*16);

    mmul(cl::EnqueueArgs( queue,
                          cl:::NDRange(N,N), cl:::NDRange(16,16)),
        N, d_A, d_B, d_C, Awrk, Bwrk);

    cl::copy(queue, d_C, begin(h_C), end(h_C));
}
```

### Blocked matrix multiply: Host

- **Setup local memory with blocks of A and B (16 by 16) that should fit in local memory.**
- **One work-item per element of the C matrix organized into 16 by 16 blocks.**
Matrix multiplication performance

- Matrices are stored in global memory.

<table>
<thead>
<tr>
<th>Case</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential C (not OpenCL)</td>
<td>887</td>
</tr>
<tr>
<td>C(i,j) per work-item, all global</td>
<td>3,926</td>
</tr>
<tr>
<td>Block oriented approach using local mem</td>
<td>119,305</td>
</tr>
</tbody>
</table>

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

The CuBLAS SGEMM provides an effective measure of peak achievable performance on the GPU. CuBLAS performance = 283,366 MFLOPS

Third party names are the property of their owners.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.
USING OPENCL FOR REAL RESEARCH
Molecular Docking in Bristol

BUDE (Bristol University Docking Engine) is one of the fastest and most accurate molecular docking codes in the world.

BUDE is being used to find new drug targets for influenza, malaria, Alzheimer's, Emphysema, Insulin signalling and more.

Molecular Docking in Bristol

Performance portable molecular docking with BUDE.

CloverLeaf: Peta→Exascale hydrodynamics mini-app

- CloverLeaf is a bandwidth-limited, structured grid code
- Solves the compressible Euler equations, which describe the conservation of energy, mass and momentum in a system
- Optimised parallel versions exist in OpenMP, MPI, OpenCL, OpenACC, CUDA and Co-Array Fortran
“On the performance portability of structured grid codes on many-core computer architectures”, S.N. McIntosh-Smith, M. Boulton, D. Curran and J.R. Price, ISC, Leipzig, June 2014. DOI: 10.1007/978-3-319-07518-1_4
CloverLeaf sustained bandwidth

“On the performance portability of structured grid codes on many-core computer architectures”, S.N. McIntosh-Smith, M. Boulton, D. Curran and J.R. Price, ISC, Leipzig, June 2014. DOI: 10.1007/978-3-319-07518-1_4
Lattice Boltzmann (LBM)

- A versatile approach for solving incompressible flows based on a simplified gas-kinetic description of the Boltzmann equation (used for CFD etc)
- Ports well to most parallel architectures
- We targeted one of the most widely used variants, D3Q19-BGK

D3Q19-BGK LBM

• To update a cell, need to access 19 of the 27 surrounding cell values in the 3D grid

Methodology

• Developed a code that was efficient but not over complicated

• "Identical" versions in OpenCL and CUDA
  – Single precision grid $128^3$ (~2m grid points, 304 MBytes)
  – The OpenCL three dimensional work-group size was fixed at $(128,1,1)$ for all OpenCL runs on all devices
  – Same arrangement for CUDA version

• The OpenMP code was as close as possible to the OpenCL/CUDA versions

• Ensured the OpenMP code was being vectorised by the compiler (latest Intel icc)
Performance results for $128^3$

Single precision results
Performance results for $128^3$

OpenCL single precision results

- K20c: 76%
- M2090: 57%
- GTX 780 Ti: 73%
- GTX 680: 65%
- HD 7970: 66%
- S10000 1GPU: 73%
- S10000 2GPU: 61%
- R9 290X: 64%
- Xeon Phi SE10P: 67%
- Xeon E5-2687: 29%

Reg. spill/fill: 80%
So perf. portable, but is it fast?

• On an Nvidia K20, our best $128^3$ D3Q19-BGK LBM single precision performance in OpenCL was 1,110 MLUPS

• In the literature, the fastest quoted results are ~1,000 MLUPS (Januszewski and Kostur's *Sailfish* program) and 982 MLUPS (Mawson and Revell)

• Our results are a 13% improvement over Mawson-Revell and a 10% improvement over Januszewski-Kostur

SOME CONCLUDING REMARKS
Conclusions

• OpenCL has *widespread* industrial support

• OpenCL defines a platform-API/framework for *heterogeneous computing*, not just GPGPU or CPU-offload programming

• OpenCL has the potential to deliver *portably performant code*; but it has to be used correctly

• The latest *C++ and Python APIs* makes developing OpenCL programs much simpler than before

• OpenCL is the *only* parallel programming standard that enables mixing task parallel and data parallel code in a single program and load balancing across a *wide variety of parallel hardware*. *It's fun to use too!!!
OpenCL-related things

• OpenCL’s Standard Portable Intermediate Representation (SPIR)
  – Based on LLVM’s Intermediate Representation (IR)
  – Makes interchangeable front- and back-ends straightforward

• OpenCL 2.0
  – Released Nov 2013, only partial implementations so far
  – Lots of other improvements

• SYCL
  – Single source C++ higher level parallel programming in OpenCL
  – http://www.khronos.org/opencl/sycl/

• For the latest news on SPIR and new OpenCL versions see:
  – http://www.khronos.org/opencl/
OpenCL resources

• Khronos website
  – https://www.khronos.org/opencl/

• Annual OpenCL workshop, IWOCL
  – http://iwocl.org
  – In May each year (in Boston for 2015)

• HandsOnOpenCL training course online
  – http://handsonopencl.github.io
  – Dozens of exercises and solutions in C, C++ and Python
  – Includes CUDA to OpenCL tutorial
OpenCL books

OpenCL Programming Guide:
Aaftab Munshi, Benedict Gaster, Timothy G. Mattson and James Fung, 2011

Heterogeneous Computing with OpenCL
Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry and Dana Schaa, 2011
I'm hiring!

• Looking for a postdoc for a 3 year post on the prestigious FP7 Mont Blanc project
  – Can we build an Exascale supercomputer from European technologies, such as ARM-based CPUs and GPUs?
  – Post to look at software fault tolerance techniques for Exascale computing

• Also looking for PhD students

• Email me at simonm at cs.bris.ac.uk