OpenCL 2.0, OpenCL SYCL & OpenMP 4
Open Standards for Heterogeneous Parallel Computing

Ronan Keryell

AMD
Performance & Application Engineering
Sunnyvale, California, USA

07/02/2014
Present and future: heterogeneous...

- Physical limits of current integration technology
  - Smaller transistors
  - More and more transistors 😊
  - More leaky: static consumption 😞
  - Huge surface dissipation 😞
    - Impossible to power all the transistors all the time ⇒ “dark silicon”
    - Specialize transistors for different tasks
    - Use hierarchy of processors (“big-little”...)
    - Use myriads of specialized accelerators (DSP, ISP, GPU, cryptoprocessors, IO...)

- Moving data across chips and inside chips is more power consuming than computing
  - Memory hierarchies, NUMA (Non-Uniform Memory Access)
  - Develop new algorithms with different trade-off between data motion and computation

- Need new programming models & applications
  - Control computation, accelerators, data location and choreography
  - Rewrite most power inefficient parts of the application 😊

OpenCL 2.0, OpenCL SYCL & OpenMP 4
Software

• Automatic parallelization
  ▶ Easy to start with
  ▶ Intractable issue in general case
  ▶ Active (difficult) research area for 40+ years
  ▶ Work only on well written and simple programs 😞
  ▶ Problem with parallel programming: current bad shape of existing sequential programs... 😞

• At least can do the easy (but cumbersome) work
New parallel languages

- Domain-Specific Language (DSL) for parallelism & HPC
- Chapel, UPC, X10, Fortress, Erlang...
- Vicious circle
  - Need to learn new language
  - Need rewriting applications
  - Most of // languages from last 40 years are dead

New language acceptance ↓ ↓
New libraries
- Allow using new features without changing language
  - Linear algebra BLAS/Magma/ACML/PetSC/Trilinos/..., FFT...
  - MPI, MCAPI, OpenGL, OpenCL...
- Some language facilities not available (metaprogramming...)
- Drop in approach for application-level libraries
• **Pragma(tic) approach**
  - Start from existing language
  - Add distribution, data sharing, parallelism... hints with `#pragma`
  - OpenMP, OpenHMPP, OpenACC, XMP (back to HPF)...  

• Portability
Software

- New concepts in existing object-oriented language
  - Domain Specific Embedded Language (DSeL)
  - Abstract new concepts in classes
  - // STL, C++AMP, OpenCL SYCL...

- Full control of the performance
Software

- Operating system support
  - Avoid moving data around
  - Deal with NUMA memory, cache and processor affinity
  - Provide user-mode I/O & accelerator interface
  - Provide virtual memory on CPU and accelerators
  - HSA...

A good answer will need a mix of various approaches

OpenCL 2.0, OpenCL SYCL & OpenMP 4
Host threads launch computational *kernels* on *accelerators*

https://www.khronos.org/opencl
Execution model

OpenCL 2.0, OpenCL SYCL & OpenMP 4
Memory model
Share Virtual Memory (SVM)

3 variations...

Coarse-Grained memory buffer SVM

- Sharing at the granularity of regions of OpenCL buffer objects
  - `clSVMAlloc()`
  - `clSVMFree()`
- Consistency is enforced at synchronization points
- Update views with `clEnqueueSVMMap()`, `clEnqueueSVMUnmap()`, `clEnqueueMapBuffer()` and `clEnqueueUnmapMemObject()` commands
- Similar to non-SVM but allows shared pointer-based data structures
Share Virtual Memory (SVM)

Fine-Grained memory buffer SVM

- Sharing at the granularity of individual loads/stores into bytes within OpenCL buffer memory objects
- Consistency guaranteed only at synchronization points
- Optional OpenCL atomics to provide fine-grained consistency
  - No need to use previous ...Map()/. . .Unmap()
Share Virtual Memory (SVM)

**Fine-Grained system SVM à la C(++)11**

- Sharing occurs at the granularity of individual loads/stores into bytes occurring **anywhere within the host memory**
  - Allow normal memory such as `malloc()`
- Loads and stores may be cached so consistency is guaranteed only at synchronization points
- Optional OpenCL atomics to provide fine-grained consistency

New pointer `__attribute__((nosvm))`
Lambda expressions with Block syntax

- From Mac OS X’s Grand Central Dispatch, implemented in Clang

```c
int multiplier = 7;
int (^myBlock)(int) = ^(int num) {
    return num*multiplier;
};
printf("%d\n", myBlock(3)); // prints 21
```

- By-reference closure but const copy for automatic variable
- Equivalent in C++11

```c
auto myBlock = [=] (int num) {
    return num*multiplier;
};
```
Device-side enqueue

- OpenCL 2 allows nested parallelism
- Child kernel enqueued by kernel on a device-side command queue
- Out-of-order execution
- Use events for synchronization
- No kernel preemption \(\Rightarrow\) Continuation-passing style! 😊

[en.wikipedia.org/wiki/Continuation-passing_style](en.wikipedia.org/wiki/Continuation-passing_style)
Device-side enqueue

```c
// Find and start new jobs
kernel void evaluate_work(...) {
    /* Check if more work needs to be performed, 
       for example a tree or list traversal */
    if (check_new_work(...)) {
        /* Start workers with the right //ism on default 
           queue only after the end of this launcher */
        enqueue_kernel(get_default_queue(),
                       CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                       ndrange_1D(compute_size(...)),
                       ^{ real_work(...); });
    }
}
```
// Cross-recursion example for dynamic //ism
kernel void real_work(...) {
    // The real work should be here
    [...]
    /* Queue a single instance of evaluate_work() to device default queue to go on recursion */
    if (get_global_id(0) == 0) {
        /* Wait for the *current* kernel execution before starting the *new one* */
        enqueue_kernel(get_default_queue(),
                        CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                        ndrange_1D(1),
                        ^{ evaluate_work(...); });
    }
}
Collective work-group operators

- Operation involving all work-items inside a work-group
  - `int work_group_all(int predicate)`
  - `int work_group_any(int predicate)`
  - `gentype work_group_broadcast(gentype a, size_t id_x...)`
  - `gentype work_group_reduce_op(gentype x)`
  - `gentype work_group_scan_exclusive_op(gentype x)`
  - `gentype work_group_scan_inclusive_op(gentype x)`

Subgroups

- Represent real execution of work-items inside work-groups
  - 1-dimensional
  - *wavefront* on AMD GPU, *warp* on nVidia GPU
  - There may be more than 1 subgroup/work-group...

- Coordinate `uint get_sub_group_id()`,
  `uint get_sub_group_local_id()`,
  `uint get_sub_group_size()`, `uint get_num_sub_groups()`...

- `void sub_group_barrier(...)`

- Collective operators `sub_group_reduce_op()`,
  `sub_group_scan_exclusive_op()`,
  `sub_group_scan_inclusive_op()`, `sub_group_broadcast()`...
Pipe

- Efficient connection between kernels for stream computing
- Ordered sequence of data items
- One kernel can write data into a pipe
- One kernel can read data from a pipe

```c
cl_mem clCreatePipe(cl_context context,
                     cl_mem_flags flags,
                     cl_uint pipe_packet_size,
                     cl_uint pipe_max_packets,
                     const cl_pipe_properties *props,
                     cl_int *errcode_ret)
```

- Kernel functions to read/write/test packets
Other improvements in OpenCL 2

- MIPmaps (*multum in parvo* map): textures at different LOD (level of details)
- Local and private memory initialization (*à la* `calloc()`)
- Read/write images `__read_write`
- Interoperability with OpenGL, Direct3D...
- Images (support for 2D image from buffer, depth images and standard IEC 61996-2-1 sRGB image)
- Linker to use libraries with `clLinkProgram()`
- Generic address space `__generic`
- Program scope variables in global address space
- C11 atomics
- Clang blocks (∼ C++11 lambda in C)
- `int printf(constant char * restrict format, ...)` with vector extensions
- Kernel-side events & profiling
- On-going Open Source implementations (AMD on HSA...)
OpenMP 4

- Target (virtual) shared memory machines
- Add some directives (#pragma...) in existing language to help compiler
  - Parallelism
  - Data sharing with weak consistency
- Warning: If no directive, no parallelism used (a priori)
- Warning Warning Warning Directive ≡ sworn declaration
- Support C/C++/Fortran by most compilers
- Also used for other languages (Python with Pythran...)
- Unification of previous vendor #pragma in 1997
- OpenMP 4 (2013) supports heterogeneous accelerators!

http://openmp.org/wp/openmp-specifications
Thread model

- Parallel execution based on fork/join

```c
#pragma omp parallel
for (i = 0; i < N; i++)
    neat_stuff(i);
```
Task programming

```c
#include <stdio.h>
int main() {
    int x = 1;
    // Start threads
    #pragma omp parallel
    // But only execute following block in a single thread
    #pragma omp single
    {
        // Start statement in a new task
        #pragma omp task shared(x) depend(out: x)
        x = 2;
        // Start statement in another task
        #pragma omp task shared(x) depend(in: x)
        printf("x/uni2423=/uni2423%d\n", x);
    }
    return 0;
```
Task programming (II)

- Can deal with normal, anti- and output dependencies
SIMD loops

- Allow vector execution in SIMD

```c
#pragma omp simd
for(int i = 0; i < N; ++i)
    x[i] = y[i] + z[i];
```

- Can limit parallelism to deal with loop-carried dependencies

```c
#pragma omp simd safelen(10)
for(int i = 10; i < N; ++i)
    x[i] = x[i - 10] + y[i];
```
SIMD version of functions

- Provide also a vector function

```c
#pragma omp declare simd uniform(v) \ 
    linear(i) notinbranch
void init(double array[N], int i, double v) {
    array[i] = v;
}

// [...]

double a[N];
double v = random123();
#pragma omp simd
for(int i = 0; i < N; ++i)
    init(a, i, v);
```
Loops with threads + SIMD execution

// Execute next block in multi-threaded way
#pragma parallel if(N > 1000)
{
    // [...]
#pragma omp for simd
    for(int i = 0; i < N; ++i)
        x[i] = y[i] + z[i];

} // End of the threads
Execution on a target device

- Possible to off-load some code to a device

```c
/* Off-load computation on accelerator
   if enough work, or keep it on host */
#pragma target if(N > 1000)
   // The loop on the device is executed in parallel
#pragma omp parallel for
for(int i = 0; i < N; ++i)
  x[i] = y[i] + z[i];
```

- Data are moved between host and target device by OpenMP compiler
Device data environment

- Execution of distributed-memory accelerators
  - Need sending parameters
  - Need getting back results
- Allow mapping host data to target device

```c
#pragma omp target data map(to: v1[N:M], v2[:M-N]) \ 
map(from: p[N:M])
{
    #pragma omp target
    #pragma omp parallel for
    for (int i = N; i < M; i+)
    
        p[i] = v1[i] * v2[i - N];
}
```
Execution with work-groups

float dotprod(int N, float B[N], float C[N],
              int block_size,
              int num_teams, int block_threads) {
    float sum = 0;
    #pragma omp target map(to: B[0:N], C[0:N])
    #pragma omp teams num_teams(num_teams) \
              thread_limit(block_threads) \
              reduction(+:sum)
    #pragma omp target distribute
      // Scheduled on the master of each team
    for (int i0 = 0; i0 < N; i0 += block_size)
        #pragma omp parallel for reduction(+:sum)
          // Executed on the threads of each team
        for (int i = i0; i < min(i0+block_size, N); ++i)
            sum += B[i]*C[i];
    return sum;
}
Other OpenMP 4 features

- Affinity control of threads/processor
- SIMD functions
- Cancellation points
- Generalized reductions
- Taskgroups
- Atomic swap
- C/C++ array syntax for array sections in clauses
- `OMP_DISPLAY_ENV` to display current ICV values
- Open Source implementation with GCC 4.9 (on-host target), Clang/LLVM on-going,...
OpenCL SYCL goals

- Ease of use
- Single source programming model
  - SYCL source compiled for host and device(s)
- Development/debugging on host
- Programming interface that data management and error handling
- C++ features available for OpenCL
  - Enabling the creation of higher level programming models and C++ templated libraries based on OpenCL
- Portability across platforms and compilers
- Providing the full OpenCL feature set and seamless integration with existing OpenCL code
- High performance

Puns explained for French speakers

- OpenCL SPIR (spear: lance, pointe)
- OpenCL SYCL (sickle: faucille)
Complete example of matrix addition

```cpp
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

constexpr size_t N = 2;
constexpr size_t M = 3;
using Matrix = float[N][M];

int main() {
    Matrix a = {{ 1, 2, 3 }, { 4, 5, 6 }};
    Matrix b = {{ 2, 3, 4 }, { 5, 6, 7 }};
    Matrix c;
```
Complete example of matrix addition (II)

```c++
#include <CL/Sycl.hpp>

namespace cl {

class mat_add {
public:
    void operator()(id<2> i) {
        // Perform matrix addition
    }
};

}

int main() {
    // Create a queue to work on
    queue myQueue;

    // Define buffers for matrices A, B, and C
    buffer<float, 2> A { a, range<2> { N, M } };
    buffer<float, 2> B { b, range<2> { N, M } };
    buffer<float, 2> C { c, range<2> { N, M } };

    // Create command group
    command_group (myQueue, [&] () {
        auto ka = A.get_access<access::read>();
        auto kb = B.get_access<access::read>();
        auto kc = C.get_access<access::write>();

        parallel_for(range<2> { N, M },
                     kernel_lambda<class mat_add>((id<2> i) {
                         kc[i] = ka[i] + kb[i];
                     }));
    }); // End of our commands for this queue

    // End scope, so wait for the queue to complete
}
```
Complete example of matrix addition

```c
return 0;
}
```
Hierarchical parallelism

```cpp
const int size = 10;
int data[size];
const int gsize = 2;
buffer<int> my_buffer { data, size };
```
Hierarchical parallelism

command_group(my_queue, [&]() {
    auto in = my_buffer.get_access<access::read>();
    auto out = my_buffer.get_access<access::write>();
    parallel_for_workgroup(nd_range<>(range<>(size),
                                   range<>(gszie)),
        kernel_lambda<class hierarchical>
            ([=](group<> grp) {
                std::cerr << "Gid=" << grp.get(0) << std::endl;
                parallel_for_workitem(grp, [=](item<1> tile) {
                    std::cerr << "id_" << tile.get_local().get(0) << " uni2423=" << tile.get_global()[0] << std::endl;
                    out[tile] = in[tile] * 2;
                });
            });
};
OpenCL SYCL road-map

http://www.khronos.org/opencl/sycl

- GDC (Game Developer Conference), March 2014
  - Released a provisional specification to enable feedback
  - Developers can provide input into standardization process
  - Feedback via Khronos forums

- Next steps
  - Full specification, based on feedback
  - Khronos test suite for implementations
  - Release of implementations

- Implementation
  - CodePlay
    - http://www.codeplay.com/portal/introduction-to-sycl

- Prototype in progress
  - triSYCL https://github.com/amd/triSYCL → Join us!
SYCL summary

- Like C++AMP but with OpenCL/OpenGL/... interoperability
  - OpenCL data types and built-in functions available
  - Possible to optimize some parts in OpenCL
- Host “fall-back” mode
- Single source & C++11 even in kernel (with usual restrictions)
  - generic kernel templates
- Errors through C++ exception handling
- Event handling through event class
- SYCL buffers are more abstract than OpenCL buffers
  - Data can reside on several accelerators
- `command_group` allows asynchronous task graphs à la StarPU through accessor declarations
Conclusion

- Heterogeneous computing \(\sim\) Rewriting applications
  - Applications are to be refactored regularly anyway...
- Entry cost...
  - \(\exists\) New tools allowing smooth integration in existing language
  - Can mix several approaches such as OpenMP + OpenCL + MPI
- Exit cost! 😊
  - Use Open Standards backed by Open Source implementations
  - Be locked or be free!
- Mobile computing is pushing!

OpenCL 2.0, OpenCL SYCL & OpenMP 4
Table of content

1. OpenCL 2
   - Outline
     - Architecture model
     - Execution model
     - Memory model
     - Share Virtual Memory (SVM)
     - Lambda expressions with Block syntax
     - Device-side enqueue
     - Collective work-group operators
     - Subgroups
     - Pipe
     - Other improvements in OpenCL 2

2. OpenMP 4
   - Outline
     - OpenMP 4
     - Thread model

3. OpenCL SYCL
   - Outline
     - OpenCL SYCL goals
     - Complete example of matrix addition
     - Hierarchical parallelism
     - OpenCL SYCL road-map
     - SYCL summary

4. Conclusion
   - Outline
     - Conclusion
   You are here!