Memory Hierarchy Visibility in Parallel Programming Languages

ACM SIGPLAN Workshop on Memory Systems Performance and Correctness MSPC 2014 Keynote

Dr. Paul Keir - Codeplay Software Ltd.

45 York Place, Edinburgh EH1 3HP

Fri 13th June, 2014

Overview

- Codeplay Software Ltd.
- Trends in Graphics Hardware
- GPGPU Programming Model Overview
  - Segmented-memory GPGPU APIs
  - GPGPU within Graphics APIs
  - Non-segmented-memory GPGPU APIs
  - Single-source GPGPU APIs
- Khronos SYCL for OpenCL
- Conclusion

Codeplay Software Ltd.

- Incorporated in 1999
- Based in Edinburgh, Scotland
- 34 full-time employees
- Compilers, optimisation and language development
  - GPU, NUMA and Heterogeneous Architectures
  - Increasingly Mobile and Embedded CPU/GPU SoCs
- Commercial partners include:
  - Qualcomm, Movidius, AGEIA, Fixstars
- Member of three 3-year EU FP7 research projects:
  - Peppher (Call 4), CARP (Call 7) and LPGPU (Call 7)
- Sony-licensed PlayStation®3 middleware provider
- Contributing member of Khronos group since 2006
- A member of the HSA Foundation since 2013

Correct and Efficient Accelerator Programming (CARP)

"The CARP European research project aims at improving the programmability of accelerated systems, particularly systems accelerated with GPUs, at all levels."

- Industrial and Academic Partners:
  - Imperial College London, UK
  - ENS Paris, France
  - ARM Ltd., UK
  - Realeyes OU, Estonia
  - RWTHA Aachen, Germany
  - Universiteit Twente, Netherlands
  - Rightware OY, Finland
- `carpproject.eu`

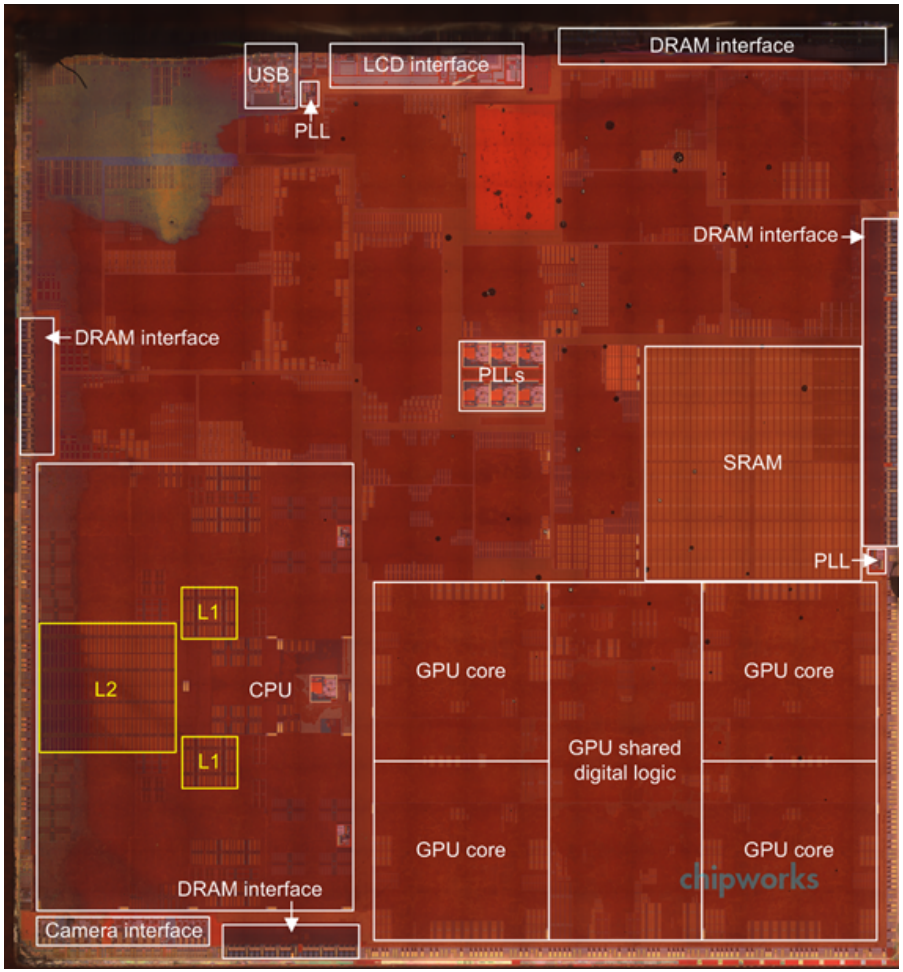
Low-power GPU (LPGPU)

"The goal of the LPGPU project is to analyze real-world graphics and GPGPU workloads on graphics processor architectures, by means of measurement and simulation, and propose advances in both software and hardware design to reduce power consumption and increase performance."

- Industrial and Academic Partners:
  - TU Berlin, Germany
  - Geomerics Ltd., UK
  - AiGameDev.com KG, Austria
  - Think Silicon EPE, Greece
  - Uppsala University, Sweden
- `lpgpu.org`

“Rogue” GPU Footprint on the Apple A7

- A GPU is most commonly a system-on-chip (SoC) component
- Trend is for die proportion occupied by the GPU to increase



Apple A7 floorplan courtesy of Chipworks

Canonical GPGPU Data-Parallel Thread Hierarchy

- Single Instruction Multiple Threads (SIMT)
- Memory latency is mitigated by:
  - launching many threads; and
  - switching warps/wavefronts whenever an operand isn't ready

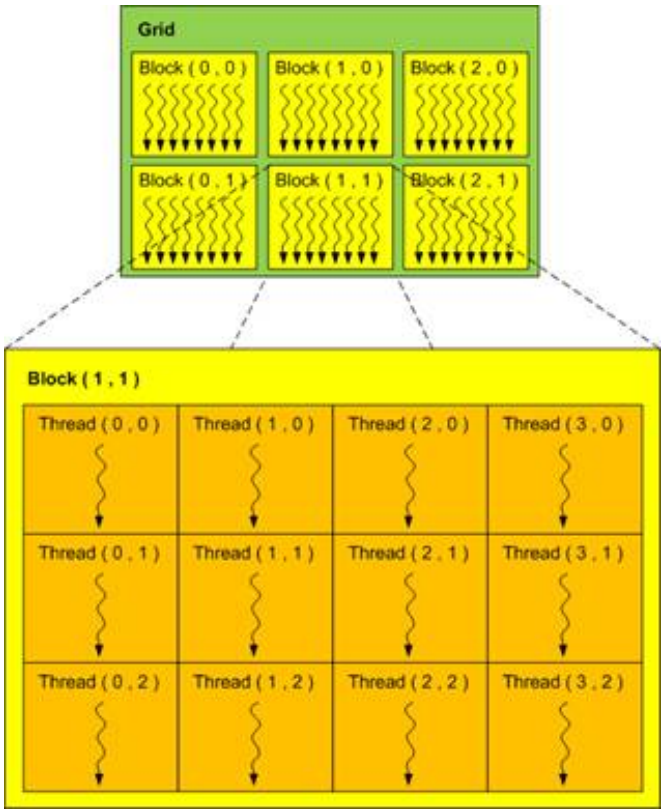


Image: `http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm`

Canonical GPGPU Data-Parallel Memory Hierarchy

- Registers and local memory are unique to a thread
- Shared memory is unique to a block
- Global, constant, and texture memories exist across all blocks.
- The scope of GPGPU memory segments:

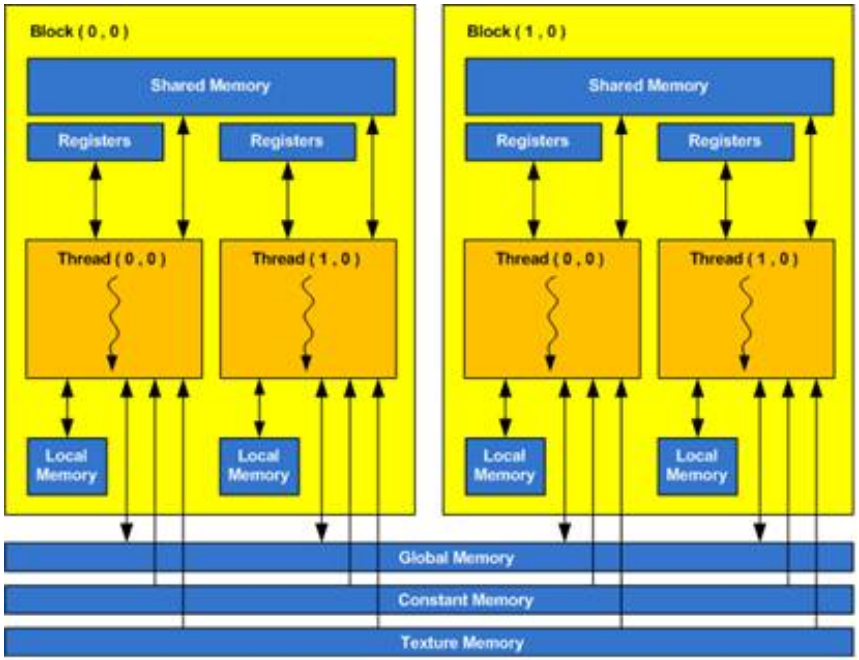


Image: `http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm`

Segmented-memory GPGPU APIs

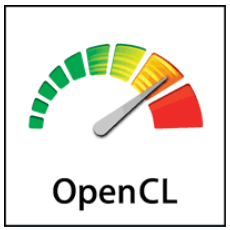
CUDA (Compute Unified Device Architecture)

- NVIDIA's proprietary market leading GPGPU API
  - Released in 2006
- A single-source approach, and an extended subset of C/C++
- The programmer defines C functions; known as *kernels*
  - When called, kernels are executed N times in parallel
  - ...by N different CUDA threads
  - Informally, an SIMT execution model
- Each thread has a unique thread id; accessible via `threadIdx`

```
__global__
void vec_add(float *a, const float *b, const float *c)
{
  uint id = blockIdx.x * blockDim.x + threadIdx.x;
  a[id] = b[id] + c[id];
}
```

OpenCL (Open Computing Language)

- Royalty-free, cross-platform standard governed by Khronos
- Portable parallel programming of heterogeneous systems
- Memory and execution model similar to CUDA
- OpenCL C kernel language based on ISO C99 standard
  - Source distributed with each application
  - Kernel language source compiled at runtime
  - 4 address spaces: `global`; `local`; `constant`; and `private`
- OpenCL 2.0: SVM; device-side enqueue; uniform pointers




```
kernel
void vec_add(global        float *a,
             global const float *b,
             global const float *c)
{
  size_t id = get_global_id(0);
  a[id] = b[id] + c[id];
}
```

OpenCL SPIR

- Khronos Standard Portable Intermediate Representation
- A portable LLVM-based non-source distribution format
- SPIR driver in OpenCL SDKs from Intel and AMD (beta)

```
define spirkrnl void @vec_add(
  float addrspace(1)* nocapture %a,
  float addrspace(1)* nocapture %b,
  float addrspace(1)* nocapture %c) nounwind
{
  %1 = call i32 @get_global_id(i32 0)

  %2 = getelementptr float addrspace(1)* %a, i32 %1
  %3 = getelementptr float addrspace(1)* %b, i32 %1
  %4 = getelementptr float addrspace(1)* %c, i32 %1

  %5 = load float addrspace(1)* %3, align 4
  %6 = load float addrspace(1)* %4, align 4
  %7 = fadd float %5, %6

  store float %7, float addrspace(1)* %2, align 4
  ret void
}
```

GPGPU within Graphics APIs

GPGPU within established Graphics APIs

Direct Compute in DirectX 11 HLSL (2008)

- Indices such as the `uvec3`-typed `SV_DispatchThreadID`
- Variables declared as `groupshared` reside on-chip
- Group synchronisation via:
  - `GroupMemoryBarrierWithGroupSync()`

Compute Shaders in OpenGL 4.3 GLSL (2012)

- Built-ins include the `uvec3` variable `gl_GlobalInvocationID`
- Variables declared as `shared` reside on-chip
- Group synchronisation via:
  - `memoryBarrierShared()`

AMD Mantle, and Microsoft DirectX 12 will soon also be released

Apple iOS 8 - Metal

- Can specify both graphics and compute functions
- Built-in vector and matrix types; e.g. `float3x4`
- 3 function qualifiers: `kernel`, `vertex` and `fragment`
  - A function qualified as *A* cannot call one qualified as *B*
  - `local` data is supported only by `kernel` functions
  - 4 address spaces: `global`; `local`; `constant`; and `private`
- Resource attribute qualifiers using C++11 attribute syntax
  - e.g. `buffer(n)` refers to nth host-allocated memory region
  - Attribute qualifiers like `global_id` comparable to Direct Compute's `SV_DispatchThreadID`

```
kernel
void vec_add(global       float *a [[buffer(0)]],
             global const float *b [[buffer(1)]],
             global const float *c [[buffer(2)]],
             uint               id [[global_id]])
{
  a[id] = b[id] + c[id];
}
```

Non-segmented-memory GPGPU APIs

OpenMP

- Cross-platform standard for shared memory parallelism
- Popular in High Performance Computing (HPC)
- A single-source approach for C, C++ and Fortran
- Makes essential use of compiler pragmas
- OpenMP 4: SIMD; user-defined reductions; and *accelerators*
- No address-space support from the type system

```
void vec_add(int n,
             float *a, float const *b, float const *c)
{
#pragma omp target teams map(  to:a[0:n])          \
                         map(from:b[0:n],c[0:n])
#pragma omp distribute parallel for
  for (int id = 0; id < n; ++id)
    a[id] = b[id] + c[id];
}
```

Google RenderScript for Android

- Runtime determines where a kernel-graph executes
  - e.g. Could construct the gaussian function $y_i = e^{-x_i^2}$ as:

```
mRsGroup = new ScriptGroup.Builder(mRS)
  .addKernel(sqrID).addKernel(negID).addKernel(expID)
  .addConnection(aType,sqrID,negID)
  .addConnection(aType,negID,expID).create();
```

- A C99-based kernel language with no local memory/barriers
- Emphasis for Renderscript is performance portability

```
#pragma version(1)
#pragma rs java_package_name(com.example.test)

float __attribute__((kernel))
vec_add(float b, float c)
{
  return b + c;
}
```

MARE (Multicore Asynchronous Runtime Environment)

- A C++ library-based approach for parallel software
  - No kernel language; no local memory
- Available for Android, Linux and Windows
  - Optimised for the Qualcomm Snapdragon™ platform
- A parallel patterns library: `pfor_each`, `pscan`, `transform`
- Task based: with dependencies forming a dynamic task graph
- Shared Virtual Memory (SVM) support from software

```
auto hello = mare::create_task([]{printf("Hello ");});
auto world = mare::create_task([]{printf("World!");});

hello >> world;

mare::launch(hello);
mare::launch(world);

mare::wait_for(world);
```

Heterogeneous System Architecture (HSA) Foundation

- HSA aims to improve GPGPU programmability
- Applications create data structures in a unified address space
- Founding members:
  - AMD, ARM, Imagination, MediaTek, Qualcomm, Samsung, TI
- HSAIL is a virtual machine and intermediate language
- Register allocation completed by the high-level compiler
- Unified memory addressing...but *seven* memory *segments*:
  - `global, readonly, group, kernarg`
  - `spill, private, arg`
  - ...the latter three typically not end-user specified
- Memory operations can optionally specify a segment
  - e.g. `ld_group_f32 $d1 $d0`
  - No explicit segment: use flat addressing
- Barrier operations also take a segment: e.g. `barrier_fgroup`

Vector Addition in HSA

- `workitemabsid_u32` provides the work-item absolute ID
- `$s2` holds the final result; then stored to `[$s1]`

```
kernel &vec_add(kernarg_u32 %arg_val0,
                kernarg_u32 %arg_val1,
                kernarg_u32 %arg_val2)
{
@vec_add_entry:
  workitemabsid_u32 $s0, 0;
  ld_kernarg_u32    $s1, [%arg_val2];
  ld_kernarg_u32    $s2, [%arg_val1];
  ld_kernarg_u32    $s3, [%arg_val0];
  shl_u32           $s0, $s0, 2;        // s0=id*sizeof(float)
  add_u32           $s2, $s2, $s0;
  ld_global_f32     $s2, [$s2];
  add_u32           $s3, $s3, $s0;
  ld_global_f32     $s3, [$s3];
  add_f32           $s2, $s3, $s2;
  add_u32           $s1, $s1, $s0;
  st_global_f32     $s2, [$s1];
  ret;
};
```

Single-source GPGPU APIs

Accelerated Massive Parallelism (C++ AMP)

- Microsoft-backed open standard for heterogeneous compute
- Builds on C++11 with two language extensions:
  - Function qualifier: `restrict`
  - Storage class: `tile_static`
- Note that `restrict` is required on all device functions

```
void vec_add(int n,
             float *A, const float *B, const float *C)
{
  array_view<int       > a(n,A);
  array_view<const int> b(n,B), c(n,C);
  parallel_for_each(a.extent,
    [&](index<1> id) restrict(amp)
  {
    a[id] = b[id] + c[id];
  });
}
```

(Before) Unified Memory Access in CUDA 6

```
int main(int argc, char *argv[]) {
  int *d_a, *d_b, *d_c;
  cudaMalloc((void **) &d_a, 1<<24);
  cudaMalloc((void **) &d_b, 1<<24);
  cudaMalloc((void **) &d_c, 1<<24);
  int *a = (int *)malloc(1<<24);
  int *b = (int *)malloc(1<<24);
  int *c = (int *)malloc(1<<24);

  cudaMemcpy(d_b,b,1<<24,cudaMemcpyHostToDevice);
  cudaMemcpy(d_c,c,1<<24,cudaMemcpyHostToDevice);

  vec_add<<<512>>>(d_a,d_b,d_c);

  cudaMemcpy(a,d_a,1<<24,cudaMemcpyDeviceToHost);

  free(a); free(b); free(c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
}
```

- cudaMallocManaged replaces cudaMalloc

(After) Unified Memory Access in CUDA 6

```
int main(int argc, char *argv[]) {
  int *d_a, *d_b, *d_c;
  cudaMallocManaged((void **) &d_a, 1<<24);
  cudaMallocManaged((void **) &d_b, 1<<24);
  cudaMallocManaged((void **) &d_c, 1<<24);









  vec_add<<<512>>>(d_a,d_b,d_c);
  cudaDeviceSynchronise();



  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
}
```

- Each pointer can access both the host and the device

Khronos SYCL for OpenCL

- Simplified software porting for existing parallel applications
- Code reuse, through sharing of host and device code
- Generic algorithms through C++ template meta-programming
- A foundation for higher-level programming models
- Host execution fallback if OpenCL device is unavailable
  - `www.khronos.org/opencl/sycl`

A Single-Source Model

...and a *Shared*-Source Model

- Existing C++ compiler processes the host sections of the code
- *Extended* C++ device compiler processes the device sections
  - Currently outputs OpenCL SPIR bitcode (based on LLVM)
- Call graph is *duplicated* for all devices targeted
  - So, a single object or datatype may be used in both contexts
- SYCL 1.2 targets OpenCL 1.2 devices, so no:
  - function pointers; virtual methods; recursion;
  - exception handling; or run-time type information
- OpenCL code and C++ code may be used together
- The C++ preprocessor can be used to select the best code
  - e.g. to guard between sections compiled for device; or host

SYCL Foundations

```
#include <SYCL/sycl.hpp>

int main(int argc, char *argv[])
{
  cl::sycl::queue q;
  return 0;
}
```

- The SYCL command queue contains a list of tasks
- These tasks are administered by the command thread
- Tasks can run on either host or device
- The queue will select default OpenCL devices; and error handling

Lambda Functions in C++11

- A Haskell lambda expression:

```
\x -> x+y
```

- A C++11 lambda expression:

```
[=](int x) -> decltype(x) { return x+y; }
```

- An equivalent C++ function object:

```
struct lambda_like {
  int operator()(int x) { return x+y; };
  int y;
};
```

A Single Asynchronous Task

```
#include <SYCL/sycl.hpp>

using namespace cl::sycl;

int main(int argc, char *argv[])
{
  queue q;
  command_group(&q, [&]()
  {
    single_task(kernel_lambda<class MyTask>([=](){}));
  });
  return 0;
}
```

- A single task will execute on the default OpenCL device
- The stub header file is generated by the device compiler
  - Allows the kernel to be linked with the host code

Wrapping C++11 Lambdas for Portability

```
void example1()
{
  queue q;
  command_group(&q, [&]()
  {
    single_task(kernel_lambda<class MyTask>([=](){}));



  });
}
```

- Each lambda function has a unique type
- C++11 lambda function is 1st argument of `kernel_lambda`
- Naming class need only be declared; in C++11 can be inline

Wrapping C++11 Lambdas for Portability

```
void example1()
{
  queue q;
  command_group(&q, [&]()
  {
    single_task(kernel_lambda<class MyTask>([=]()
    {
      // kernel code
    }));
  });
}
```

- Each lambda function has a unique type
- C++11 lambda function is 1st argument of `kernel_lambda`
- Naming class need only be declared; in C++11 can be inline

The SYCL Command Group

```
void example1()
{
  queue q;
  command_group(&q, [&]()
  {
    single_task(kernel_lambda<class MyTask>([=]()
    {
      // kernel code
    }));
  });
}
```

- A command group defines multiple task/data parallel kernels
- Command group captures variables within scope *by reference*
- The kernel captures closure variables *by value*
  - Allowing the runtime to pass variables to and from the device

SYCL Buffers and Accessors

```
void example2()
{
  queue q;
  float f = 0.0f;
  buffer<float> data(&f, 1);
  command_group(&q, [&]()
  {
    auto kdata = data.get_access<access::write>();
    single_task(kernel_lambda<class MyTask>([=]()
    {
      kdata[0] = 3.142f;
    }));
  });
  cout << f << '\n';    // 3.142
}
```

- An SYCL `buffer` object allows for data reuse
- A single `buffer` object can serve multiple `accessor` objects
- Also possible to express data dependencies through sub-buffers

Execution Modes in SYCL

- SYCL supports both task and data-parallelism
- Data-parallel operational modes:
  - Basic data-parallelism:
    - `parallel_for` with a `range<int>` argument
  - Workgroup data-parallelism:
    - `parallel_for` with an `nd_range<int>` argument
  - Hierarchical data-parallelism:
    - `parallel_for_workgroup` *and* `parallel_for_workitem`
    - ...with `nd_range<int>` and `group<int>`
- One task-parallel mode:
  - Task parallelism obtained via `single_task`; shown earlier
- Can also provide a kernel as an OpenCL C string

SYCL Basic Data-Parallelism

```
void example3()
{
  queue q;
  float f[64];
  buffer<float> data(f, 64);
  command_group(&q, [&]()
  {
    auto kdata = data.get_access<access::write>();

    parallel_for(range<3>(4, 4, 4),
      kernel_lambda<class Bdp>([=](item id) {
      kdata[id] = id.get_global_id(0);
    }
  });
}
```

- A simple range executes over a range of n dimensions

SYCL Basic Data-Parallelism

```
void example4()
{
  queue q;
  float f[64];
  buffer<float> data(f, 64);
  command_group(&q, [&]()
  {
    auto kdata = data.get_access<access::write>();

    parallel_for({4,4,4},
      kernel_lambda<class Bdp>([=](item id) {
      kdata[id] = id.get_global_id(0);
    }
  });
}
```

- A simple range executes over a range of n dimensions
  - ...can be specified using a C++11 initializer list

SYCL Basic Data-Parallelism

```
void example5()
{
  queue q;
  buffer<float> data(64);
  command_group(&q, [&]()
  {
    auto kdata = data.get_access<access::write>();

    parallel_for({4,4,4},
      kernel_lambda<class Bdp>([=](item id) {
      kdata[id] = id.get_global_id(0);
    }
  });
}
```

- A simple range executes over a range of n dimensions
  - ...can be specified using a C++11 initializer list
  - ...and we can let the buffer object allocate memory

Workgroup Data-Parallelism

```
void example6()
{
  queue q;
  buffer<float> data(64);
  command_group(&q, [&]()
  {
    auto kdata = data.get_access<access::write>();

    parallel_for({4,4,4},{2,2,2},
      kernel_lambda<class Wdp>([=](item id) {
      __local float data[8];
      kdata[id] = id.get_global_id(0);
      barrier(CLK_LOCAL_MEM_FENCE); // for example
    }
  });
}
```

- OpenCL C syntax is permitted within the SYCL kernel

Hierarchical Data-Parallelism

```
void example7()
{
  queue q;
  buffer<float> data(64);
  command_group(&q, [&]()
  {
    auto  in_access = data.access< read_only>();
    auto out_access = data.access<write_only>();

    parallel_for_workgroup({4,4,4},{2,2,2},
      kernel_lambda<class Hdp>([=](group group)
    {
      __local float data[8];
      parallel_for_workitem(group, [=](item id)
      {
        out_access[id] = in_access[id] * 2;
      });
    }));
  });
}
```

OpenCL C Kernels

```
const char *src = R"(
__kernel void q(__global int *p) {
  p[get_global_id(0)] = get_global_id(0); })";
void example8()
{
  buffer<float> data(64);
  default_selector sel(opencl_c);
  context ctx(&sel);
  queue   q(&ctx);
  program p(src, &ctx);
  kernel *k = p.compile_kernel_by_name("q");

  command_group(&q, [&]()
  {
    auto p = data.get_access<access::read_write>();
    set_kernel_arg(k,p);
    parallel_for({64}, k);
  });
}
```

High Performance Computing

- Message Passing Interface (MPI)
- MapReduce and Hadoop
- OmpSs from BSC
- Partitioned Global Address Space Languages
  - CoArrays in Fortran 2008
  - X10 and Chapel
  - XcalableMP; Titanium
- Functional HPC
  - Single Assignment C
  - GpH; Data Parallel Haskell; and Repa

Conclusion

- A time of rapid innovation for graphics hardware & integration
- Driven by a market hunger for realtime graphical fidelity
- Numerous emerging graphics and GPGPU APIs & languages
- New industry standardisation effort from the HSA Foundation
- An ambition to combine programmability and performance
- Khronos SYCL for OpenCL:
  - `http://www.khronos.org/opencl/sycl`