Part 1

COMPILING OPENCL KERNELS

Shipping OpenCL Kernels

- OpenCL applications rely on just-in-time (JIT) compilation in order to achieve portability
- Shipping source code with applications can be an issue for commercial users of OpenCL
- There are a few ways to try and hide your OpenCL kernels from end users

Encrypting OpenCL Source

- One approach is to encrypt the OpenCL source, and decrypt it at runtime just before passing it to the OpenCL driver
- This could achieved with a standard encryption library, or by applying a simple transformation such as Base64 encoding
- This prevents the source from being easily read, but it can still be retrieved by intercepting the call to `clCreateProgramWithSource()`
- Obfuscation could also be used to make it more difficult to extract useful information from the plain OpenCL kernel source

Precompiling OpenCL Kernels

- OpenCL allows you to retrieve a binary from the runtime after it is compiled, and use this instead of loading a program from source next time the application is run
- This means that we can precompile our OpenCL kernels, and ship the binaries with our application instead of the source code

Precompiling OpenCL Kernels

- Retrieving the binary (single device):

```
// Create and compile program
program = clCreateProgramWithSource(context, 1, &kernel_source, NULL, &err);
clBuildProgram(program, 1, &device, "", NULL, NULL);

// Get compiled binary from runtime
size_t size;
clGetProgramInfo(program, device, CL_PROGRAM_BINARY_SIZES, 0, &size, NULL);
unsigned char *binary = malloc(size);
clGetProgramInfo(program, device, CL_PROGRAM_BINARIES, size, binary, NULL);

// Then write binary to file
…
```

- Loading the binary

```
// Load compiled program binary from file
…

// Create program using binary
program = clCreateProgramWithBinary(context, 1, &device,
                                    size, &binary, NULL, &err);
err = clBuildProgram(prog, 1, &device, "", NULL, NULL);
```

Precompiling OpenCL Kernels

- These binaries are only valid on the devices for which they are compiled, so we potentially have to perform this compilation for every device we wish to target
- A vendor might change the binary definition at any time, potentially breaking our shipped application
- If a binary isn't compatible with the target device, an error will be returned either during `clCreateProgramWithBinary()` or `clBuildProgram()`

Portable Binaries (SPIR)

- Khronos have produced a specification for a **S**tandard **P**ortable **I**ntermediate **R**epresentation
- This defines an LLVM-based binary format that is designed to be portable, allowing us to use the same binary across many platforms
- Not yet supported by all vendors

Stringifying Kernel Source

- We usually load our OpenCL kernel source code from file(s) at runtime
- We can make things easier by using a script to convert OpenCL source files into string literals defined inside header files
- This script then becomes part of the build process:

```
foo.h: foo.cl
    ./stringify_ocl foo.cl
```

Stringifying Kernel Source

- This script makes use of SED to escape special characters and wrap lines in quotation marks

```
#!/bin/bash

IN=$1
NAME=${IN%.cl}
OUT=$NAME.h

echo "const char *"$NAME"_ocl =" >$OUT
sed -e 's/\\/\\\\/g;s/"/\\"/g;s/^/"/;s/$/\\n"/' \
  $IN >>$OUT
echo ";" >>$OUT
```

Stringifying Kernel Source

Before stringification:

```
kernel void vecadd(
  global float *a,
  global float *b,
  global float *c)
{
  int i =
    get_global_id(0);
  c[i] = a[i] + b[i];
}
```

After stringification:

```
const char *vecadd_ocl =
"kernel void vecadd(\n"
"  global float *a,\n"
"  global float *b,\n"
"  global float *c)\n"
"{\n"
"  int i =\n"
"    get_global_id(0);\n"
"  c[i] = a[i] + b[i];\n"
"}\n"
;
```

Generating Assembly Code

- Can be useful to inspect compiler output to see if the compiler is doing what you think it's doing
- On NVIDIA platforms the 'binary' retrieved from `clGetProgramInfo()` is actually PTX, their abstract assembly language
- On AMD platforms you can add `–save-temps` to the build options to generate `.il` and `.isa` files containing the intermediate representation and native assembly code
- Intel provide an offline compiler which can generate LLVM/SPIR or x86 assembly

Kernel Introspection

- We can query a program object for the names of all the kernels that it contains:

```
clGetProgramInfo(…,
      CL_PROGRAM_NUM_KERNELS, …);
clGetProgramInfo(…,
      CL_PROGRAM_KERNEL_NAMES, …);
```

- We can also query information about kernel arguments (OpenCL 1.2):

```
clGetKernelInfo(…, CL_KERNEL_NUM_ARGS, …);
clGetKernelArgInfo(…, CL_KERNEL_ARG_*, …);
```

(the program should be compiled using the `-cl-kernel-arg-info` option)

Kernel Introspection

- This provides a mechanism for automatically discovering and using new kernels, without having to write any new host code
- Can make it much easier to add new kernels to an existing application
- Provides a means for libraries and frameworks to accept additional kernels from third parties

Separate Compilation and Linking

- OpenCL 1.2 gives more control over the build process by adding two new functions:

  ```
  clCompileProgram()
  clLinkProgram()
  ```

- This enables the creation of libraries of compiled OpenCL functions, that can be linked to multiple program objects

Compiler Options

- OpenCL compilers accept a number of flags that affect how kernels are compiled:

```
-cl-opt-disable
-cl-single-precision-constant
-cl-denorms-are-zero
-cl-fp32-correctly-rounded-divide-sqrt
-cl-mad-enable
-cl-no-signed-zeros
-cl-unsafe-math-optimizations
-cl-finite-math-only
-cl-fast-relaxed-math
```

implies

Compiler Flags

- Vendors may expose additional flags to give further control over program compilation, but these will not be portable between different OpenCL platforms
- For example, NVIDIA provide **`-cl-nv-arch`** to control which GPU architecture to target, and **`-cl-nv-maxrregcount`** to limit the number of registers used
- Some vendors support **`-O`*`n`*** flags to control the optimization level
- AMD allow additional build options to be dynamically added using an environment variable: **`AMD_OCL_BUILD_OPTIONS_APPEND`**

Metaprogramming

- We can exploit JIT compilation to embed values that are only know at runtime into kernels as compile-time constants
- In some cases this can significantly improve performance
- OpenCL compilers support the same preprocessor definition flags as GCC/Clang:

  `-Dname`

  `-Dname=value`

Example: Multiply a vector by a constant value

Passing the value as an argument

```
kernel void vecmul(
  global float *data,
  const  float  factor)
{
  int i = get_global_id(0);
  data[i] *= factor;
}
```

```
clBuildProgram(
  program, 1, &device,
  "", NULL, NULL);
```

Not known at application build time (e.g. passed as command-line argument)

Defining the value as a preprocessor macro

```
kernel void vecmul(
  global float *data)
{
  int i = get_global_id(0);
  data[i] *= factor;
}
```

```
char options[32];
sprintf(
  options, "-Dfactor=%f",
  argv[1]);
clBuildProgram(
  program, 1, &device,
  options, NULL, NULL);
```

Metaprogramming

- Can be used to dynamically change the precision of a kernel
  - Use `REAL` instead of `float/double`, then define `REAL` at runtime using OpenCL build options: `–DREAL=type`
- Can make runtime decisions that change the functionality of the kernel, or change the way that it is implemented to improve performance portability
  - Switching between scalar and vector types
  - Changing whether data is stored in buffers or images
  - Toggling use of local memory
- All of this requires that we are compiling our OpenCL sources at runtime – this doesn't work if we are precompiling our kernels or using SPIR

Part 2

DEBUGGING OPENCL APPLICATIONS

Debugging OpenCL Applications

- Debugging OpenCL programs can be very hard
- You don't always have the 'luxury' of a segmentation fault – on a GPU that might turn into an unexplainable OpenCL API error, a kernel panic, artifacts appearing on screen or no symptoms at all
- Functional errors are equally difficult to track down – you're typically running thousands of work-items concurrently
- At worst, your only debugging tool is to copy intermediate values from your kernel back to the host and inspect them there
- But with any luck you'll have a few more tools to work with

printf

- OpenCL 1.2 defines `printf` as a built-in function available within kernels
- Useful to perform quick sanity checks about intermediate values
- Remember that the kernel is potentially being executed by *lots* of work-items
  - Output order is undefined
  - Guard with `if(get_global_id(0) == …)` to inspect a specific work-item (adjust for 2D/3D)

Debugging with GDB

- GDB works with OpenCL running on the CPU with AMD® or Intel® runtimes
- Useful for stepping through kernel execution, and catching some illegal memory accesses
- Can be a bit fiddly to get working, and requires different setup instructions for each platform

Using GDB with Intel®

- Ensure you select the CPU device from the Intel® platform
- Enable debugging symbols and add the absolute path to the kernel source code when building the kernels:

```
clBuildProgram(… "-g -s /path/to/kernel.cl" …);
```

- The symbolic name of a kernel function '`kernel void foo(args)`' will just be `foo`
  - To set a breakpoint on kernel entry enter at the GDB prompt:

    ```
    break foo
    ```

  - This can only be done *after* the kernels have been built
- On Windows, this functionality is provided via a graphical user interface inside Visual Studio

Using GDB with AMD®

- Ensure you select the CPU device from the AMD® platform
- Enable debugging symbols and turn off all optimizations when building the kernels:

  ```
  clBuildProgram(… "-g -O0" …);
  ```

- The symbolic name of a kernel function '`kernel void foo(args)`' will be `__OpenCL_foo_kernel`
  - To set a breakpoint on kernel entry enter at the GDB prompt:

    ```
    break __OpenCL_foo_kernel
    ```

  - This can only be done *after* the kernels have been built
- AMD® recommend setting the environment variable `CPU_MAX_COMPUTE_UNITS=1` to ensure deterministic kernel behaviour

CodeXL

- AMD have a graphical tool called [CodeXL]
- Provides the ability to debug OpenCL kernels running on the GPU
  - Step through kernel source
  - Inspect variables across work-items and work-groups
  - Display contents of buffers and images
- Allows applications to be debugged on remote machines
- Also supports CPU and GPU profiling
  - Collecting hardware counters
  - Visualizing kernel timelines
  - Occupancy and hotspot analysis

GPUVerify

- A useful tool for detecting data-races in OpenCL programs
- Developed at Imperial College as part of the CARP project
- Uses static analysis to try to prove that kernels are free from races
- Can also detect issues with work-group divergence
- More information on the [GPUVerify Website]

```
gpuverify --local_size=64,64 --num_groups=256,256 kernel.cl
```

Oclgrind

- A SPIR interpreter and OpenCL simulator
- Developed at the University of Bristol
- Runs OpenCL kernels in a simulated environment to catch various bugs:
  - `oclgrind ./application`
  - Invalid memory accesses
  - Data-races (`--data-races`)
  - Work-group divergence
  - Runtime API errors (`--check-api`)
- Also has a GDB-style interactive debugger
  - `oclgrind -i ./application`
- More information on the [Oclgrind Website]()

Part 3

PERFORMANCE, PROFILING, AND TOOLS

Performance

```
__kernel void mmul(const int Mdim, const int Ndim, const int Pdim,
 __global float* A, __global float* B, __global float* C)
{
 int k;
 int i = get_global_id(0);
 int j = get_global_id(1);
 float tmp;

 if ( (i < Ndim) && (j <Mdim))
 {
   tmp = 0.0;
   for(k=0;k<Pdim;k++)
     tmp += A[i*Ndim+k] * B[k*Pdim+j];
   C[i*Ndim+j] = tmp;
 }
}
```

GEMM - 13 lines (From HandsOnOpenCL)

Performance

```
__kernel void mmul( const int Mdim, const int Ndim, const int Pdim,
 __global float* A, __global float* B, __global float* C,
 __local float* Bwrk)
{
  int k, j;
  int i = get_global_id(0);
  int iloc = get_local_id(0);
  int nloc = get_local_size(0);
  float Awrk[1024];
  float tmp;

  if (i < Ndim) {
    for (k = 0; k < Pdim; k++)
      Awrk[k] = A[i*Ndim+k];
    for (j = 0; j < Mdim; j++) {
      for (k = iloc; k < Pdim; k += nloc)
        Bwrk[k] = B[k*Pdim+j];
      barrier(CLK_LOCAL_MEM_FENCE);
      tmp = 0.0f;
      for (k = 0; k < Pdim; k++)
        tmp += Awrk[k] * Bwrk[k];
      C[i*Ndim+j] = tmp;
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
}
```

GEMM - 26 lines (From HandsOnOpenCL)

Performance

```
    //
    // Load A values
    //
    %IF(%ITEMY) #pragma unroll %ITEMY
    for(uint i = 0; i < (%V * (%ITEMY_BY_V)) /* PANEL * ITEMY/V */; i++)
    {
        const uint yiterations = %ITEMY_BY_V;
        uint c = (i / yiterations);
        uint r = (i % yiterations);

        #ifndef M_TAIL_PRESENT
        AVAL[c][r] = %VLOAD(0, (&A[(rowA + r*threadsY*(V)) + (ACOL + c)*lda]) );
        #else
        AVAL[c][r] = %VLOAD(0, (&A[((rowA + r*threadsY*(V)) % MV) + (ACOL + c)*lda]) );
        #endif

        #ifdef COMPLEX
        AVALEVEN[c][r] = AVAL[c][r].even;
        AVALODD[c][r] = AVAL[c][r].odd;
        #endif
    }
}

%IF(%V) #pragma unroll %V
for(uint panel=0; panel<(%V); panel++)
{
    %IF(%ITEMY_BY_V) #pragma unroll %ITEMY_BY_V
    for(uint i=0; i<(%ITEMY_BY_V); i++)
    {
        %IF(%ITEMX_BY_V) #pragma unroll %ITEMX_BY_V
        for(uint j=0; j<(%ITEMX_BY_V); j++)
        {
            const int CX = j * (%V);

            #ifndef COMPLEX
            %VFOR_REAL
            {
                CVAL[i][CX + %VFORINDEX] = mad(AVAL[panel][i],
                                 BVAL[j][panel]%VFORSUFFIX,
                                               CVAL[i][CX + %VFORINDEX]);
            }
            #else
```

```
            #ifndef __SYMM_DIAGONAL__
                #ifndef N_TAIL_PRESENT
                    SCALAR = B[ACOL*ldb + (colB + bcol)];
                #else
                    SCALAR = B[ACOL*ldb + ((colB + bcol) % NV)];
                #endif
            #else
                #ifndef N_TAIL_PRESENT
                    SCALAR = SYMM_SCALAR_LOAD(B, N, ldb, (colB + bcol), ACOL );
                #else
                    SCALAR = SYMM_SCALAR_LOAD(B, N, ldb, ((colB + bcol) % NV), ACOL);
                #endif
            #endif

            #ifdef CONJUGATE_B
                %CONJUGATE(1, SCALAR);
            #endif
            BVAL[bcol] = (SCALAR);
        }
    }

    //
    // Load A values
    //
    %IF(%ITEMY_BY_V) #pragma unroll %ITEMY_BY_V
    for(uint i = 0; i < (%ITEMY_BY_V); i++) // 1 * ITEMY/V
    {
        #ifndef M_TAIL_PRESENT
        AVAL[i] = %VLOAD(0, (&A[(rowA + i*threadsY*(V)) + (ACOL)*lda]) );
        #else
        AVAL[i] = %VLOAD(0, (&A[((rowA + i*threadsY*(V)) % MV) + (ACOL)*lda]) );
        #endif
    }

    {
        %IF(%ITEMY_BY_V) #pragma unroll %ITEMY_BY_V
        for(uint i=0; i<(%ITEMY_BY_V); i++)
        {
            %IF(%ITEMX) #pragma unroll %ITEMX
            for(uint j=0; j<(%ITEMX); j++)
            {
                %VMAD(CVAL[i][j] , AVAL[i] , BVAL[j]);
            }
        }
```

GEMM – 1647 lines (From clBLAS)

Profiling

- It's hard to tell whether code will run fast just by looking at it, especially with low level OpenCL/CUDA
- Bad performance is a bug
- Problems might not be in kernels:
  - Enqueueing `clFinish` after kernel calls
  - Inappropriate work group size for architecture
  - Slow memory copying between device and host

How do we tell where the bottlenecks are?

OpenCL events

- Used for memory copying, kernel queueing, etc.

```
cl::Event prof_event;
cl_ulong start, end;
queue.enqueueNDRangeKernel(kernel,
       offset_range,
       global_range,
       local_range,
       NULL,
       prof_event);
prof_event.wait();
prof_event.getProfilingInfo(CL_PROFILING_COMMAND_START, &start);
prof_event.getProfilingInfo(CL_PROFILING_COMMAND_END, &end);
double time_taken = static_cast<double>(end-start)*1.0e-6;
```

- The simplest way to accurately time things
- Should work everywhere

Profiling tools

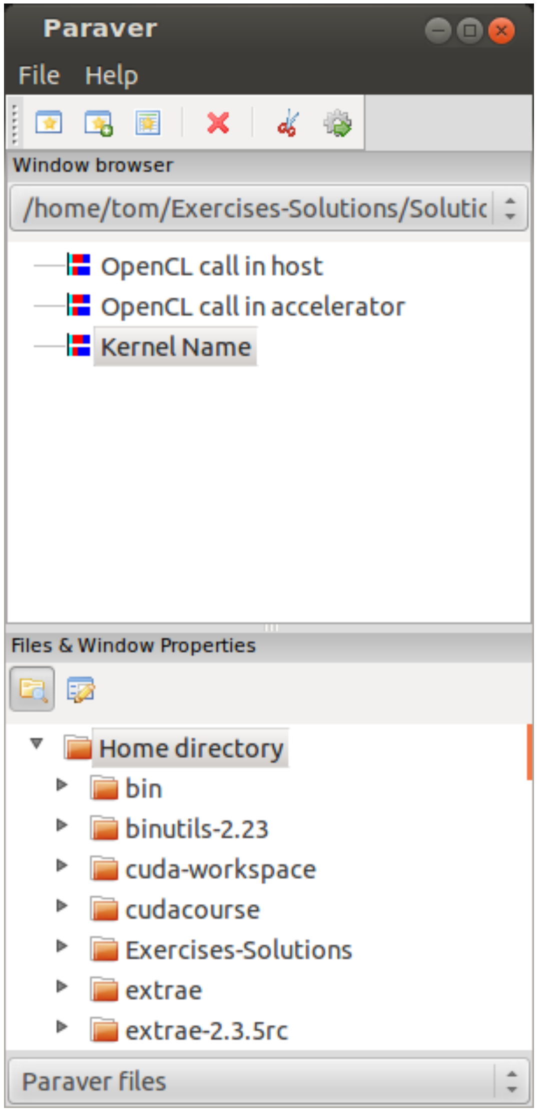
- Intel's offline compiler shows whether your kernel is being vectorised for the target device – if it can’t vectorise it, then it won’t run well!
- Intel's VTune shows memory use, parallelism, instructions taken etc. for OpenCL kernels, and has source level profiling
- Old versions of NVIDIA's nvvp show memory bandwidth, occupancy, etc.
- AMD's CodeXL provides similar functionality for AMD hardware
- ARM's DS-5 is another similar tool

Extrae and Paraver

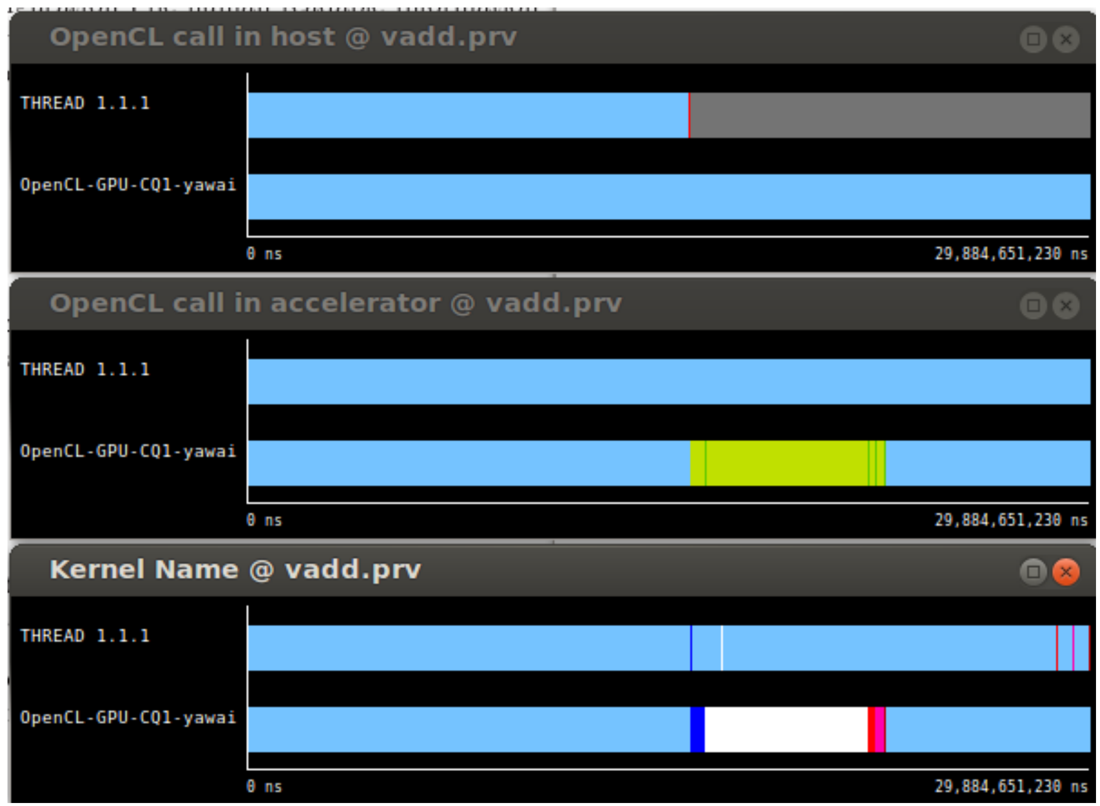
1. Extrae *instruments* your application and produces “timestamped events of runtime calls, performance counters and source code references”
   – Allows you to measure the run times of your API and kernel calls

2. Paraver provides a way to view and analyze these traces in a graphical way

Paraver

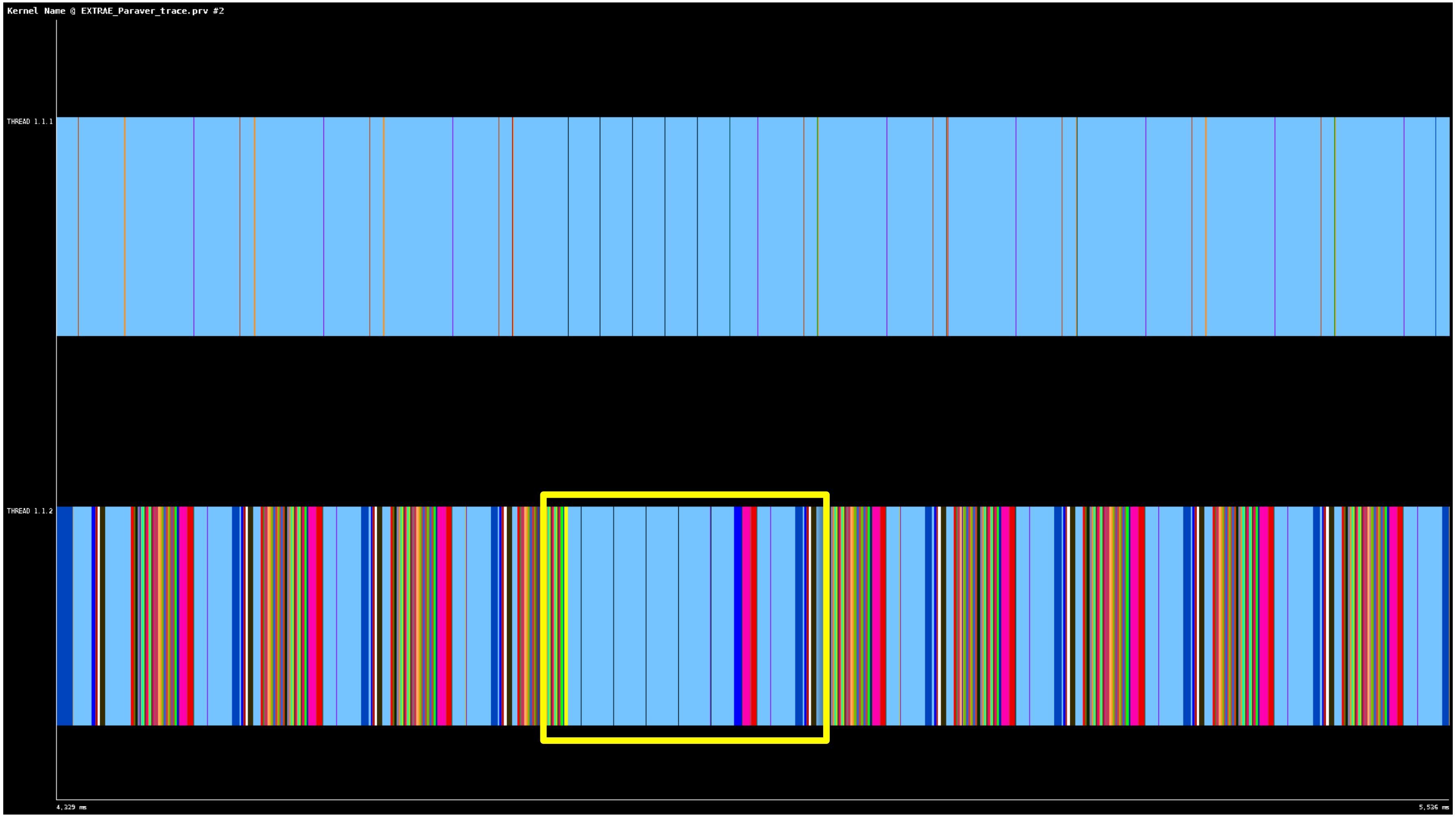


Paraver
File Help
Window browser
/home/tom/Exercises-Solutions/Solutic
OpenCL call in host
OpenCL call in accelerator
Kernel Name
Files & Window Properties
Home directory
bin
binutils-2.23
cuda-workspace
cudacourse
Exercises-Solutions
extrae
extrae-2.3.5rc
Paraver files

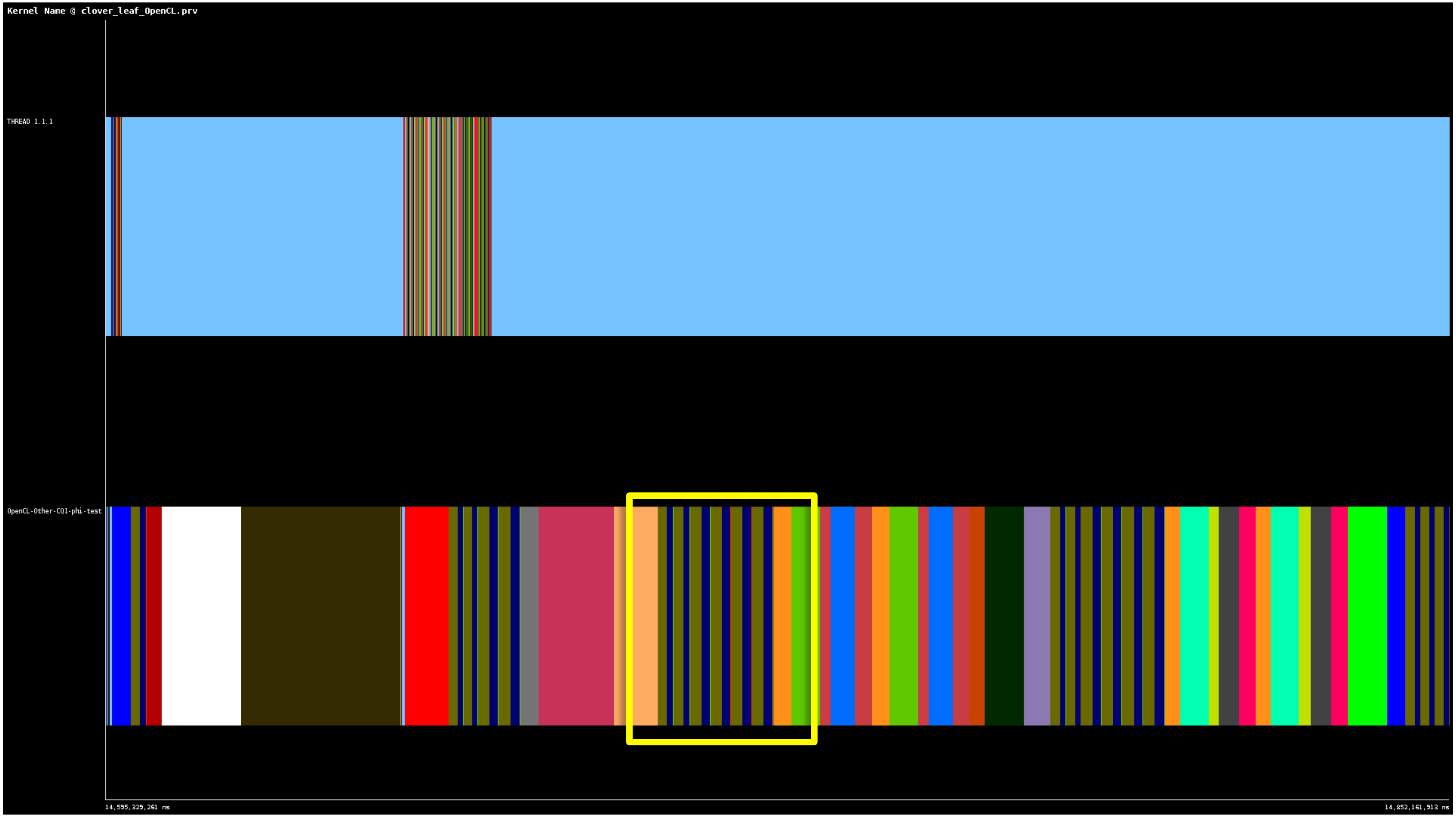


OpenCL call in host @ vadd.prv
THREAD 1.1.1
OpenCL-GPU-CQ1-yawai
0 ns
29,884,651,230 ns
OpenCL call in accelerator @ vadd.prv
THREAD 1.1.1
OpenCL-GPU-CQ1-yawai
0 ns
29,884,651,230 ns
Kernel Name @ vadd.prv
THREAD 1.1.1
OpenCL-GPU-CQ1-yawai
0 ns
29,884,651,230 ns

Paraver example



Paraver example



Exercise 1

- The exercise is a simple N-Body code
  - At each timestep, each body experiences a gravitational force from every other body in the system
  - Each work-item computes the forces acting on a single body, and updates its velocity and position
- A fully working (naïve) implementation of this code is provided as a starting point

Exercise 1

- Login to the test machines using the hostname, username and password provided to you
  - `ssh username@hostname.cs.bris.ac.uk`
  - Where `hostname` is either `yawai` (NVIDIA) or `nowai` (AMD)
- Compile and run the exercise:
  - `cd exercise`
  - `make`
  - `./nbody`
  - Make sure everything works!
- Run `./nbody --help` for a list of options
  - You can list available devices with `./nbody --list`
  - You can select a device with `./nbody --device ID`
- Familiarise yourself with the host and kernel code
- Try using the command-line profilers:
  - `COMPUTE_PROFILE=1 ./nbody` (NVIDIA)
  - `/opt/AMDAPPPROF/x86_64/sprofile -o nbody.atp -t -T -w . ./nbody` (AMD)

Exercise 1

- Experiment with some OpenCL compiler options to improve performance
- Try embedding some simulation parameters into the kernel as compile-time constants using OpenCL build options
  - This won't help for every parameter
  - This won't help on every device – try it on a few!
- Add a command-line argument (e.g. `--unroll`) to dynamically control the amount of unrolling inside the kernel (replacing the static `UNROLL_FACTOR` definition)
- An example solution will be provided
- If you have time, play around with the tools available on the test machines

Part 4

HOST-DEVICE COMMUNICATIONS

Platform discovery

- A machine may have any number of OpenCL *platforms*
- Each with their own *devices*
- Some devices may even be aliases across platforms (CPU, usually)
- How can you reliably pick your devices?

Hard coding

- Only good if you know what machine your code will always run on
- Simplest to implement
- If this is good enough, why not!

```
//get platforms
cl_platform_id platforms[2];
clGetPlatformIDs(1, platforms, NULL);

//get devices from the first platform
cl_device_id devices[3];
clGetDeviceIDs(platforms[0],
CL_DEVICE_TYPE_ALL, 3, devices, NULL);

//create context from the last device
return clCreateContext(NULL, 1,
&devices[2], NULL, NULL, NULL);
```

Selection

- Pass platform & device numbers in command line (with sane defaults)
- Much more flexible
- Needs more code..
- Also beware – cl_uint is used for device cardinality..

```
cl_context
getDevice
(int plat_num, int dev_num)
{
  //get number of platforms, devices
  cl_uint num_platforms;
  clGetPlatformIDs(0, NULL, &num_platforms);

  cl_platform_id platforms[num_platforms];
  clGetPlatformIDs(num_platforms, platforms,
NULL);

  cl_uint num_devices;
  clGetDeviceIDs(platforms[plat_num],
CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);

  cl_device_id devices[num_devices];
  clGetDeviceIDs(platforms[plat_num],
CL_DEVICE_TYPE_ALL, num_devices, devices, NULL);

  //remember: check ids are in range..
  return clCreateContext(NULL, 1,
&devices[dev_num], NULL, NULL, NULL);
}
```

Selection

- Give each platform/ device a unique number
- Pass a single argument
- Much cleaner
- But requires quite a bit more code..

```
# alternatively, in python, this
# triggers interactive device
# selection (no C required!)
pyopencl.create_some_context(True)
```

Pinned Memory

- In general, the fewer transfers you can do between host and device, the better.
- But some are unavoidable!
- It is possible to speed up these transfers, by using *pinned memory* (also called page-locked memory)
- If supported, can allow faster host <-> device communications

Pinned Memory

- A regular enqueueRead/enqueueWrite command might manage ~6GB/s
- But PCI-E Gen 3.0 can sustain transfer rates of up to 16GB/s
- So, where has our bandwidth gone?
- The operating system..
- Why? Well, when does memory get allocated?

Malloc Recap

- Consider a laptop which has 16GB of RAM.
- What is the output of the code on the right if run on this laptop?
- Bonus Question: if compiled with -m32, what will the output be?

```
#include <stdlib.h>
#include <stdio.h>

int
main
(int argc, char **argv)
{
  //64 billion floats
  size_t len    = 64 * 1024*1024*1024;

  //256GB allocation
  float *buffer = malloc(len*sizeof(float));

  if (NULL == buffer)
  {
    fprintf(stderr, "malloc failed\n");
    return 1;
  }

  printf("got ptr %p\n", buffer);
  return 0;
}
```

```
dan at srsly in ~
% gcc test.c -o test

dan at srsly in ~
% ./test
got ptr 0x7f84b0c03350
```

Malloc Recap

- We got a non-NULL pointer back..
- Both OS X and Linux will *oversubscribe* memory
- OK, so.. When will this memory actually get allocated?
- Checking the return value of malloc/calloc is useless – malloc *never* returns NULL! Really!

```
#include <stdlib.h>
#include <stdio.h>

int
main
(int argc, char **argv)
{
  //64 billion floats
  size_t len    = 64 * 1024*1024*1024;

  //256GB allocation
  float *buffer = malloc(len*sizeof(float));

  if (NULL == buffer)
  {
    fprintf(stderr, "malloc failed\n");
    return 1;
  }

  printf("got ptr %p\n", buffer);
  return 0;
}
```

Malloc Recap

- This program does not actually allocate any memory.
- We call malloc, but we never use it!

```
#include <stdlib.h>
#include <stdio.h>

int
main
(int argc, char **argv)
{
  size_t len    = 16 * 1024*1024;

  float *buffer = malloc(len*sizeof(float));

  return 0;
}
```

Malloc Recap

- So what happens here?
- The pointer we got back, when accessed, will trigger a page fault in the kernel.
- The kernel will then allocate us some memory, and allow us to write to it.
- But how much was allocated in this code? Only 4096 bytes! (One page size)

```
#include <stdlib.h>
#include <stdio.h>

int
main
(int argc, char **argv)
{
  size_t len    = 16 * 1024*1024;

  float *buffer = malloc(len*sizeof(float));

  buffer[0] = 10.0f;

  return 0;
}
```

Malloc Recap

- 4KB pages will be allocated at a time, and can also be swapped to disk dynamically.
- In fact, an allocation may not even be contiguous..
- So, enqueueRead/enqueueWrite *must* incur an additional host memory copy!

- EnqueueWrite:
  - Copy host data into a contiguous portion of DRAM
  - Signal the DMA engines to start the transfer
- EnqueueRead:
  - Allocate contiguous portion of DRAM
  - Signal DMA engine to start transfer
  - Wait for interrupt to signal that the transfer has finished
  - Copy transferred data into memory in the host code’s address space.

- Pinned memory side-steps this issue by giving the host process *direct* access to the portions of host memory that the DMA engines read and write to.
- This results in much less time spent waiting for transfers!

- Disclaimer: Not all drivers support it, and it makes allocations much more expensive (so it would be slow to continually allocate and free pinned memory!)

Getting Pinned Memory

- OpenCL has no support for pinned memory (it's not mentioned in the OpenCL spec!)
- But NVIDIA allow pinned memory allocations via CL_MEM_ALLOC_HOST_PTR flag.
- When you allocate cl_mem object, you also allocate page-locked host memory of the same size.
- But this will not return the host pointer!
- Reading and writing data is handled by enqueueMapBuffer, which *does* return the host pointer

```
//create device buffer
cl_mem devPtrA = clCreateBuffer(
  context,
  CL_MEM_ALLOC_HOST_PTR, //pinned memory flag
  len,
  NULL, //host pointer must be NULL
  NULL
);

float *hostPtrA =
(float *) clEnqueueMapBuffer(
  queue,
  devPtrA,
  CL_TRUE, //blocking map
  CL_MAP_WRITE_INVALIDATE_REGION, //write data
  0,       //offset of region
  len,     //amount of data to be mapped
  0, NULL, NULL, //event information
  NULL     //error code pointer
);
```

Caveats

- Again, allocating pinned memory is much more expensive (about 100x slower) than regular memory, so frequent allocations will be bad for performance.
- However, frequent reads and writes will be much faster!
- Not all platforms support pinned memory. But, the above method will still work, and at least will not be any slower than regular use

Multiple Devices

- Running across multiple devices can deliver better performance (if your problem scales well)
- Remember, the cost of moving data to/from a device are much greater than normal memcpys, so avoid where possible
- There are several options for using multiple devices

Multiple Contexts

- The simplest method – just call `clCreateContext` multiple times, with a different device id.
- This is only useful if you don't need to move data between devices – clEnqueueCopyBuffer can't work with memory objects created in different contexts

Multiple Command Queues

- clCreateContext can support more than one device, although only within the same platform.
- This allows copies between devices.
- However, there must be a separate command queue for each device in the context.

OpenCL & MPI

- Using MPI, it is possible to use multiple devices.
- Typically, each MPI process gets a single device.
- This allows any number of OpenCL devices.
- However, moving memory between them can be very expensive.

Halo Exchange

- If you can split your problem up into regions, then the edges must be synchronized across devices
- OpenCL allows for copying rectangular regions of a 3D buffer with clEnqueueReadBufferRect/writeBufferRect
- This is good approach to get something working; however, in practice this method is usually quite slow
- A much better alternative is to write kernels that will pack/unpack buffer regions into contiguous chunks that can be read directly, although this is much more complicated

Exercise 2

- Improve the performance of the device-host data transfers by using pinned memory
  - You might need to experiment with different approaches to see improvements on all platforms
- An example solution will be provided

Part 5

OPTIMISATIONS

Fast Kernels

- Newcomers to OpenCL tend to try and overcomplicate code ("GPUs are hard, therefore my code must be hard!")
- Adding too many levels of indirection at the start is doomed to failure (starting off with using local memory, trying to cache data yourself)
- Modern runtimes and compilers are pretty smart!
- Start simple. But once you have something working..

Performance portability

Obviously a very large field, but some basic concepts to keep in mind:

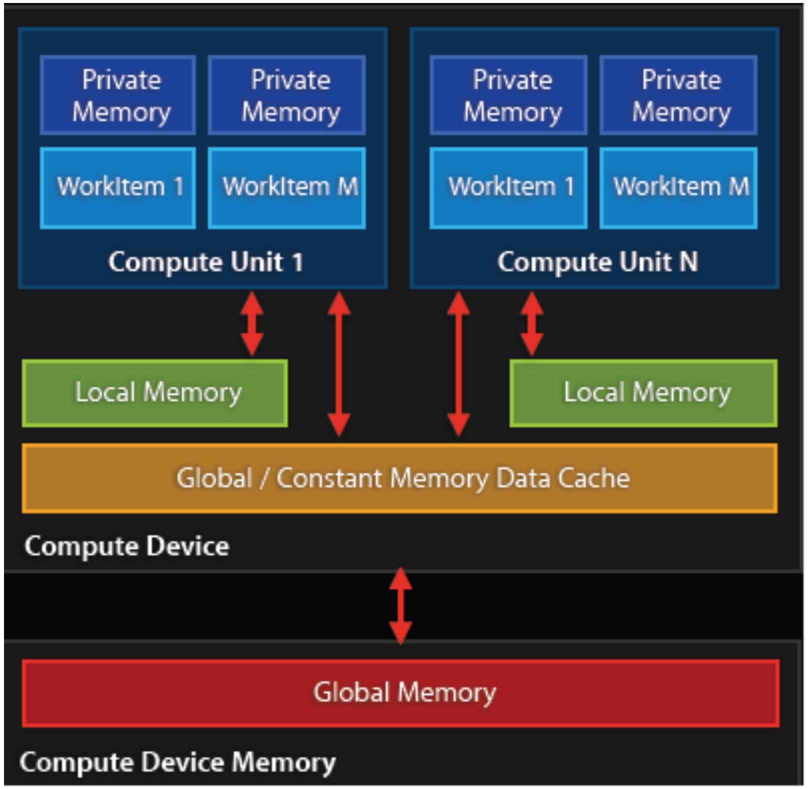
- Don’t (over-) optimise specifically for one piece of hardware
- Test on various platforms during development to make sure it actually *works* on different hardware
- Profile (events should work everywhere!)

OpenCL Memory Hierarchy

- OpenCL has 4 address spaces
- Kernels are “dumb” – data movement between address spaces will not happen automatically*
- However, manual use can sometimes improve performance (if you know something the compiler or runtime does not!)

Private Memory

- This is the default address space for variables defined in your kernel
- Memory access time is the fastest at O(1) cycles.
- But they are limited in numbers!



Private Memory

- This is the default address space for variables defined in your kernel
- Memory access time is the fastest at O(1) cycles.
- But they are limited in numbers!
- Each variable maps to a register on the device of execution
- But variables are not limited, they will be spilled into memory “somewhere” (usually local memory)
- “Occupancy” must also be considered..

```
kernel void
calc_diff
(
  global float *a,
  global float *b,
  global float *c
)
{
  //”id” is in private memory
  const int id = get_global_id(0);

  c[id] = fabs(a[id] - b[id]);
}
```

Occupancy

- NVIDIA's K40 has 128 words of memory per processor element (PE) i.e 128 registers per core.
- But, multiple work-items (threads) will be scheduled on a single PE (similar to hyperthreading)
- In fact, global memory latency is so high that multiple work-items per PE are a *requirement* for achieving a good proportion of peak performance!

Local Memory

- Local memory is the next level up from private.
- Still reasonably fast to access at O(10) cycles.
- Local memory is *shared* between work-items inside a *local workgroup*.
- Ideal use-case is when there is lots of data that gets reused amongst threads within a workgroup.
- It can be allocated either in the host, or inline in the kernel*
- When used well, can result in significant performance increases

```
kernel void
calc_something
(
  global float *a,
  global float *b,
  global float *c,

  //this local memory is set by the host
  local  float *t
)
{
  //kernels can also declare local memory
  local float tmp[128];

  //etc.
}
```

Global Memory

- Global memory is the mechanism through which your host code will communicate with the device.
- This is where data you want processed will be resident, and where output data will be written to.
- Kernel access time has *massive* latency, but high bandwidth (> 300GB/s on high-end GPUs!).
- However, latency can be hidden through *coalesced* accesses.
- That said, it's typically better to re-compute data (at the expense of private memory) than store it..!

```
size_t len = 1024*1024 * sizeof(float);
float *hostPtrA = malloc(len);

//create device buffer
cl_mem devPtrA = clCreateBuffer(
  context,          //pointer to context
  CL_MEM_READ_WRITE,//memory flags
  len,              //size of buffer (bytes)
  NULL,             //host pointer
  NULL              //error code pointer
);

clEnqueueWriteBuffer(
  queue,            //pointer to queue
  devPtrA,          //host pointer
  CL_FALSE,         //blocking write
  0,                //offset into device ptr
  len,              //number of bytes to write
  hostPtrA,         //host pointer
  0, NULL, NULL     //event list data
);
```

Coalesced Access

- As mentioned, coalesced memory accesses are key for highly performant code.
- In principle, it’s very simple, but frequently requires transposing/ transforming data on the host before sending it to the GPU
- Sometimes this is an issue of AoS vs. SoA
- Using sub buffers can help in this regard

Sub Buffers

- If you have positional data, you may be tempted to create a structure with x,y,z coordinates.
- But when it comes to running on a GPU, this strided access will be slower than contiguous access.
- `clCreateSubBuffer` allows you to create a region within a pre-existing buffer, which could ease the process of converting data to SoA format.

Those slides I
done about
coalesced
access

Constant Memory

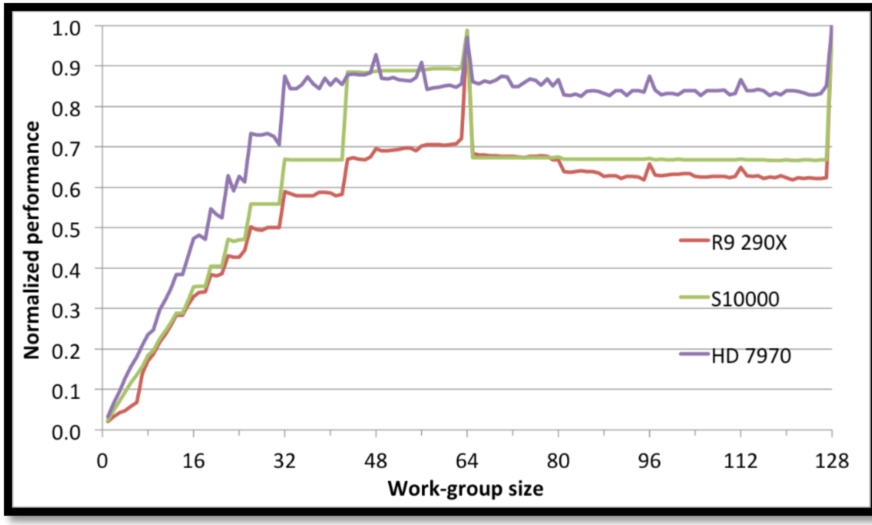
- Constant memory can be considered a store for variables that never change (i.e, are constant!)
- Setting and updating constants in memory uses the same interface as global memory, with enqueueRead/ enqueueWrite commands.
- The difference is how it is declared in the kernel
- If a device has constant memory, upon kernel execution, the data will be copied once from global.
- GPUs typically have ~64k of constant memory.

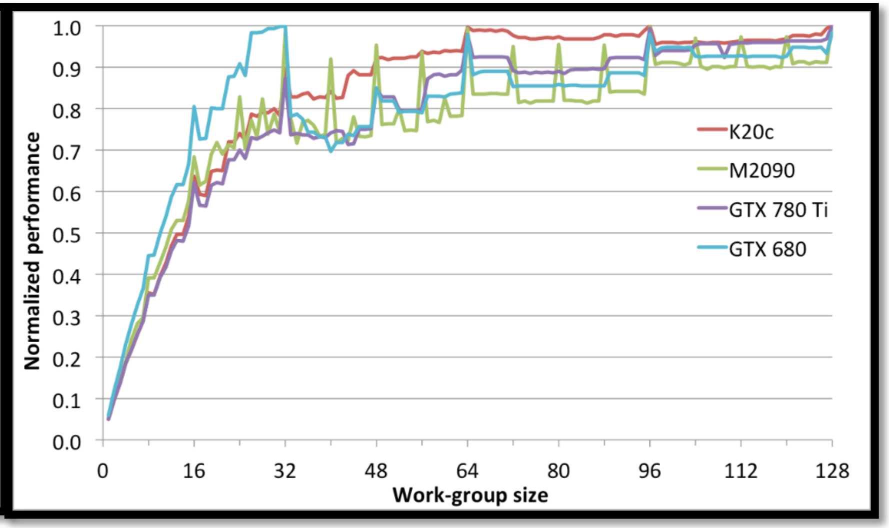
```
kernel void
calc_something
(
  global float *a,
  global float *b,
  global float *c,

  //constant memory is set by the host
  constant float *params
)
{
  //code here
}
```

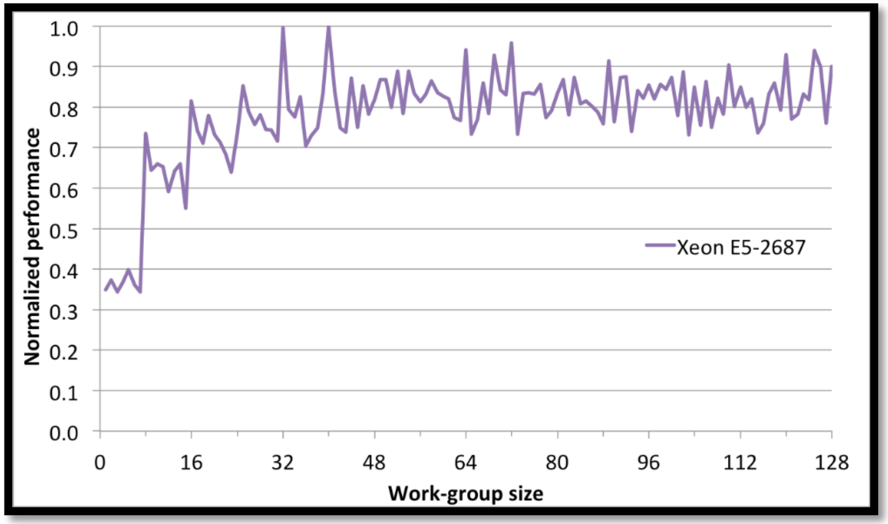
Work-groups

- 2 or 3 dimensional work-group sizes are mainly just for convenience, but do hint to the runtime what you are trying to achieve in the kernel
- Work-group sizes being a power of 2 helps on most architectures. At a minimum:
  - 8 for AVX CPUs
  - 16 for Xeon Phi
  - 32 for Nvidia
  - 64 for AMD
  - May be different on different hardware
- On Xeon Phi, try to run lots of work-groups – multiples of the number of threads available (e.g. 240 on a 5110P) is optimal, but as many as possible is good (1000+)
- NULL work-group size (cl::NullRange) might be good!

Effect of work-group sizes







Thread throttling

- Barriers between memory access-heavy kernel code sections might actually speed it up by helping the caches
- Helps temporal locality of data
- Architecture dependent

Barrier example

```
left_flux   = (xarea[THARR2D(0, 0, 1)]
    * (xvel0[THARR2D(0, 0, 1)] + xvel0[THARR2D(0, 1, 1)]
    + xvel0[THARR2D(0, 0, 1)] + xvel0[THARR2D(0, 1, 1)]))
    * 0.25 * dt * 0.5;
barrier(CLK_LOCAL_MEM_FENCE);
right_flux  = (xarea[THARR2D(1, 0, 1)]
    * (xvel0[THARR2D(1, 0, 1)] + xvel0[THARR2D(1, 1, 1)]
    + xvel0[THARR2D(1, 0, 1)] + xvel0[THARR2D(1, 1, 1)]))
    * 0.25 * dt * 0.5;
barrier(CLK_LOCAL_MEM_FENCE);
bottom_flux = (yarea[THARR2D(0, 0, 0)]
    * (yvel0[THARR2D(0, 0, 1)] + yvel0[THARR2D(1, 0, 1)]
    + yvel0[THARR2D(0, 0, 1)] + yvel0[THARR2D(1, 0, 1)]))
    * 0.25 * dt * 0.5;
barrier(CLK_LOCAL_MEM_FENCE);
top_flux    = (yarea[THARR2D(0, 1, 0)]
    * (yvel0[THARR2D(0, 1, 1)] + yvel0[THARR2D(1, 1, 1)]
    + yvel0[THARR2D(0, 1, 1)] + yvel0[THARR2D(1, 1, 1)]))
    * 0.25 * dt * 0.5;
```

Compilation hints

- When using 2 or 3 dimensional work group sizes with a local size of 1 in some dimension, consider using `get_group_id` instead of `get_global_id`
- Can specify the `reqd_work_group_size` attribute to hint to the compiler what you're going to launch it with
- As with C/C++, use the `const`/`restrict` keywords for the inputs where appropriate to make sure the compiler can optimise memory accesses (`-cl-strict-aliasing` in 1.0/1.1 as well)
- Try to use unsigned types for indexing and branching

Memory issues

- Use the `__constant` qualifier for small, read-only data items (16KB minimum, but can query to find the actual size). Some architectures might have explicit caches for this
- Strictly aligning data on power of 2 boundaries (16, 32, 64 etc) almost always helps performance

Vectorisation

- OpenCL C provides a set of vector types:
  - `type2`, `type3`, `type4`, `type8` and `type16`
  - Where `type` is any primitive data type
- Than can be convenient for representing multi-component data:
  - Pixels in an image (RGBA)
  - Atoms or points (x, y, z, mass/type)
- There are also a set of built-in geometric functions for operating on these types (`dot`, `cross`, `distance`, `length`, `normalize`)

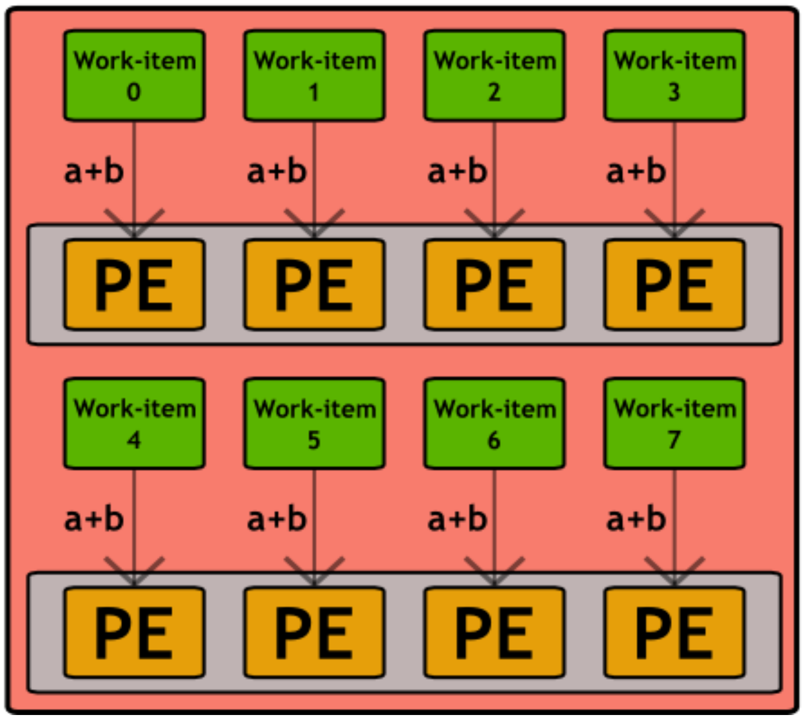
Vectorisation

- In the past, several platforms required the use of these types in order to make use of their vector ALUs (e.g. AMD's pre-GCN architectures and Intel's initial CPU implementation)
- This isn't ideal: we are already exposing the data-parallelism in our code via OpenCL's NDRange construct – we shouldn't have to do it again!
- These days, most OpenCL implementations target SIMD execution units by packing work-items into SIMD lanes – so we get the benefits of these vector ALUs for free (Intel calls this '*implicit vectorisation*')

Vectorisation

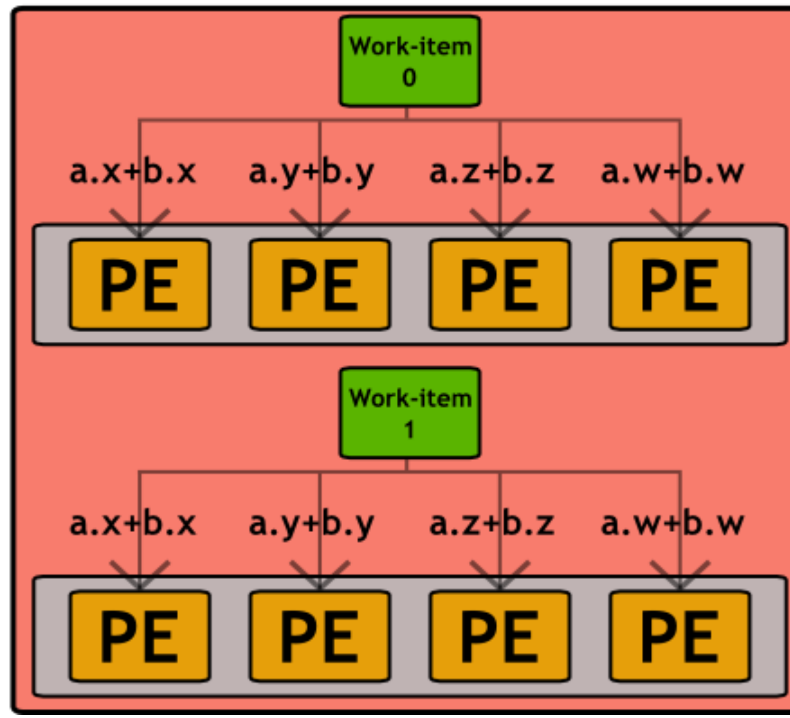
Implicit vectorisation

```
float a = ...;
float b = ...;
float c = a + b;
```



Explicit vectorisation

```
float4 a = ...;
float4 b = ...;
float4 c = a + b;
```



Vectorisation

- Unfortunately, some platforms still require explicit vectorisation, e.g.
  - ARM Mali GPUs
  - Qualcomm Adreno GPUs
- As the architectures and compilers mature, we expect to see a continued shift towards simple, scalar work-items
- You can query an OpenCL device to determine whether it prefers scalar or vector data types:

```
clGetDeviceInfo(…,
   CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT,
   …)
```

Branching

- GPUs tend not to support speculative execution, which means that branch instructions have high latency
- This latency can be hidden by switching to alternative work-items/work-groups, but avoiding branches where possible is still a good idea to improve performance
- When different work-items executing within the same SIMD ALU array take different paths through conditional control flow, we have *divergent branches*
- These are even worse: work-items will stall while waiting for the others to complete
- We can use predication, selection and masking to convert conditional control flow into straightline code and significantly improve the performance of code that has lots of conditional branches

Branching

Conditional execution

```
// Only evaluate expression
// if condition is met
if (a > b)
{
  acc += (a - b*c);
}
```

Corresponding PTX

```
setp.gt.f32}` %pred, %a, %b
@!%pred bra $endif
mul.f32 %f0, %b, %c
sub.f32 %f1, %a, %f0
add.f32 %acc, %acc, %f1
```

Selection and masking

```
// Always evaluate expression
// and mask result
temp = (a - b*c);
mask = (a > b ? 1.f : 0.f);
acc += (mask * temp);
```

Corresponding PTX

```
mul.f32 %f0, %b, %c
sub.f32 %temp, %a, %f0
setp.gt.f32 %pred, %a, %b
selp.f32 %mask, %one, %zero, %pred
mad.f32 %acc, %mask, %temp, %acc
```

Native Math Functions

- OpenCL has a large library of built-in math functions (C99 + more)
- These functions have well defined precision requirements
- Some of these functions also have native variants, which drop the precision requirements in favour of performance
- These functions start with a `native_` prefix, e.g. `native_cos`, `native_log`, `native_rqsrt`
- If you can settle for reduced precision, then these functions can significantly improve performance

Exercise 3

- Try some of these optimisations on the N-Body kernel code
- In particular, you should consider:
  - Experiment with work-group sizes
  - Caching positions in local memory (blocking)
  - Experiment with native math functions
- An example solution with all of the above applied will be provided.

Part 6

THE OPENCL ECOSYSTEM

OpenCL 2.0

- OpenCL 2.0 was ratified in Nov'13
- Brings several new features:
  - Shared Virtual Memory
  - Nested parallelism
  - Built-in work-group reductions
  - Generic address space
  - Pipes
  - C1x atomics
- Specification and headers available here
- Current beta implementations available from Intel and AMD, with more expected to follow

SPIR

- Standard Portable Intermediate Representation
- Defines an LLVM-derived IR for OpenCL programs
- Means that developers can ship portable binaries (LLVM bitcode), instead of their OpenCL source
- Also intended to be a target for other languages/ programming models (C++ AMP, SYCL, OpenACC, DSLs)
- SPIR 1.2 ratified Jan'14, SPIR 2.0 provisional available now
- Implementations available from Intel and AMD, with more on the way

SYCL

- Single source C++ abstraction layer for OpenCL
- Goal is to enable the creation of C++ libraries and frameworks that utilize OpenCL
- Can utilize SPIR to target OpenCL platform
- Supports ‘host-fallback’ (CPU) when no OpenCL devices available
- Provisional specification released Mar’14
- Codeplay and AMD working on implementations

SYCL

```
std::vector h_a(LENGTH);             // a vector
std::vector h_b(LENGTH);             // b vector
std::vector h_c(LENGTH);             // c vector
std::vector h_r(LENGTH, 0xdeadbeef); // d vector (result)
// Fill vectors a and b with random float values
int count = LENGTH;
for (int i = 0; i < count; i++) {
  h_a[i] = rand() / (float)RAND_MAX;
  h_b[i] = rand() / (float)RAND_MAX;
  h_c[i] = rand() / (float)RAND_MAX;
}
{
  // Device buffers
  buffer d_a(h_a);
  buffer d_b(h_b);
  buffer d_c(h_c);
  buffer d_r(h_d);
  queue myQueue;
  command_group(myQueue, [&]()
  {
    // Data accessors
   auto a = d_a.get_access<access::read>();
   auto b = d_b.get_access<access::read>();
   auto c = d_c.get_access<access::read>();
   auto r = d_r.get_access<access::write>();
    // Kernel
    parallel_for(count, kernel_functor([ = ](id<> item) {
      int i = item.get_global(0);
      r[i] = a[i] + b[i] + c[i];
    }));
  });
}
```

Example code from [Codeplay's SYCL tutorial]

Source level

- C/C++ API
- PyOpenCL
- PGI/CAPS OpenACC to OpenCL
- Some other languages have support now (Julia)
- Halide

Libraries

- Arrayfire (open source soon)
- Boost compute with VexCL
- ViennaCL (PETSc), PARALUTION
- clFFT/clBLAS
- Lots more

Applications

- BUDE/CloverLeaf/Rotorsim
- Science - Mont Blanc codes, GROMACS
- Desktop – Libreoffice, Adobe video processing
- Games
- etc

Links

- http://streamcomputing.eu/blog/2013-06-03/the-application-areas-opencl-can-be-used/
- http://lpgpu.org/wp/wp-content/uploads/2014/02/PEGPUM_2014_intel.pdf
- http://hgpu.org/?tag=opencl
- http://www.khronos.org/opencl/resources