Lecture 10

OPTIMIZING OPENCL PERFORMANCE

Extrae and Paraver

- From Barcelona Supercomputing Center
  - http://www.bsc.es/computer-sciences/performance-tools/trace-generation
  - http://www.bsc.es/computer-sciences/performance-tools/paraver
- Create and analyze traces of OpenCL programs
  - Also MPI, OpenMP
- Required versions:
  - Extrae v2.3.5rc
  - Paraver 4.4.5

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

Important!

- At the moment NVIDIA® GPUs support up to OpenCL v1.1 and AMD® and Intel® support v1.2
- If you want to profile on NVIDIA® devices you **must** compile Extrae against the NVIDIA headers and runtime otherwise v1.2 code will be used by Extrae internally which will cause the trace step to segfault

Installing Extrae and Paraver

- Paraver is easy to install on Linux
  - Just download and unpack the binary
- Extrae has some dependencies, some of which you'll have to build from source
  - libxml2
  - binutils-dev
  - libunwind
  - PAPI
  - MPI (optional)
- Use something like the following command line to configure before "make && make install":

```
./configure --prefix=$HOME/extrae --with-
binutils=$HOME --with-papi=$HOME --with-mpi=$HOME
--without-dyninst --with-unwind=$HOME --with-
opencl=/usr/local/ --with-opencl-libs=/usr/lib64
```

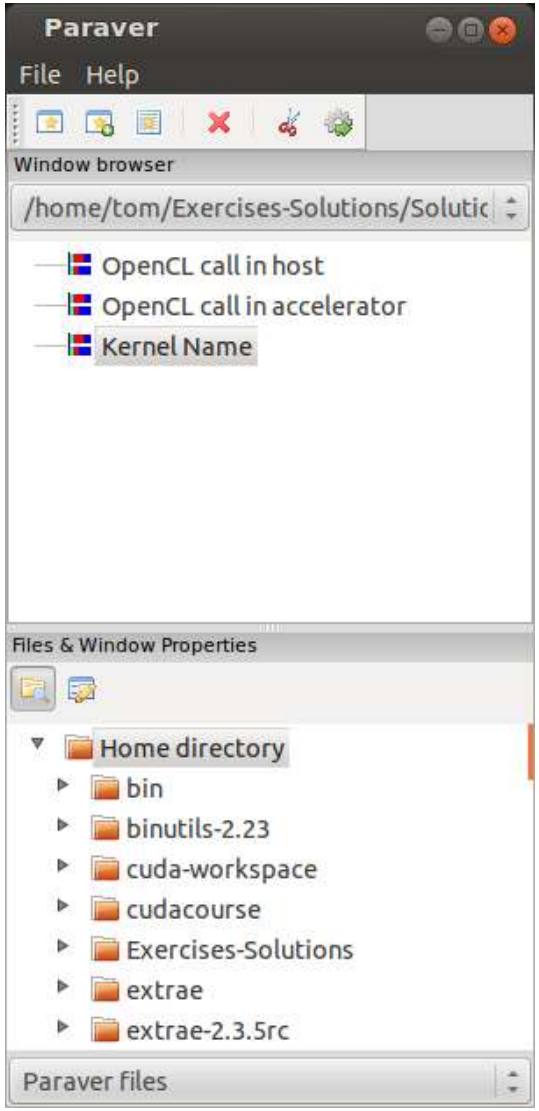
Step 1 – tracing your code

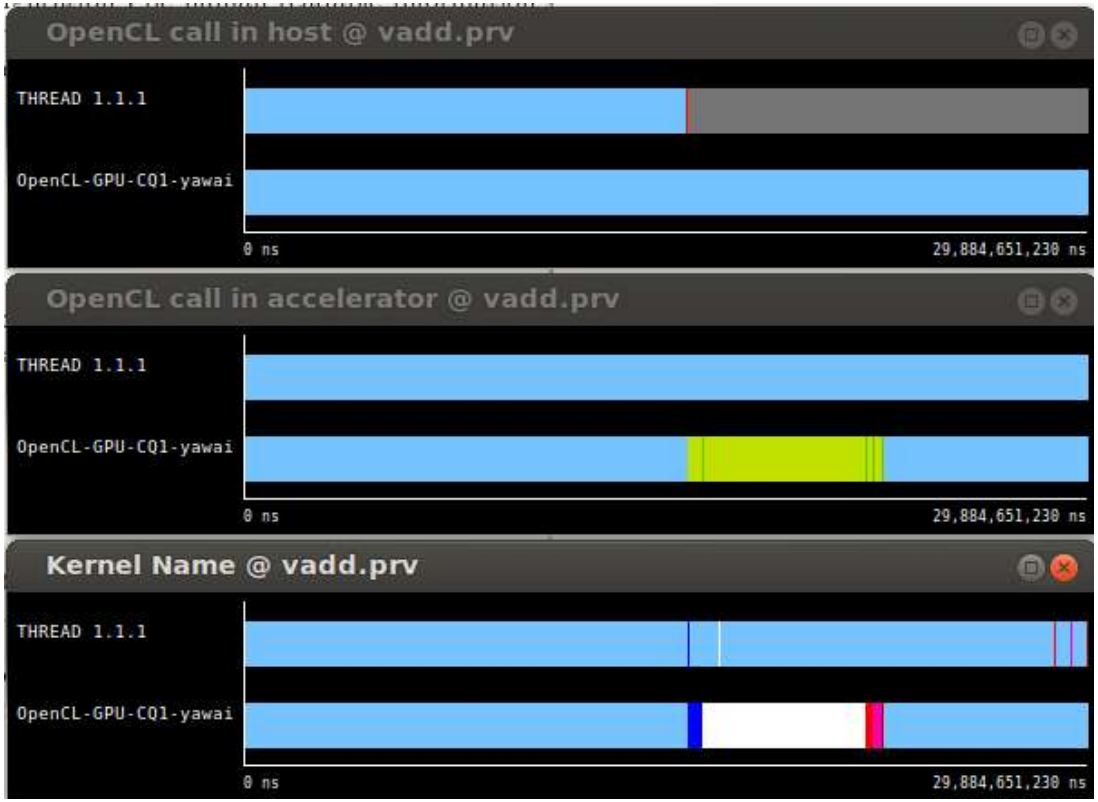
- Copy the trace.sh script from extrae/share/example/OPENCL to your project directory
  - This sets up a few environment variables and then runs your compiled binary
- Copy the extrae.xml file from the same location to your project directory
  - This gives some instructions to Extrae as to how to profile your code
  - Lots of options here – see their user guide
  - The default they provide is fine to use to begin with
- Trace!
  - **`./trace.sh ./a.out`**

Step 2 – visualize the trace

- Extrae produces a number of files
  - .prv, .pcf, .row, etc…
- Run Paraver
  - **`./wxparaver-<version>/bin/wxparaver`**
- Load in the trace
  - File -> Load Trace -> Select the .prv file
- Load in the provided OpenCL view config file
  - File -> Load configuration -> wxparaver-<version>/cfgs/OpenCL/views/opencl_call.cfg
- The traces appear as three windows
  1. OpenCL call in host - timings of API calls
  2. Kernel Name – run times of kernel executions
  3. OpenCL call in accelerator – information about total compute vs memory transfer times

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

Usage Tips

- Show what the colours represent
  - Right click -> Info Panel
- Zoom in to examine specific areas of interest
  - Highlight a section of the trace to populate the timeline window
- Tabulate the data – numerical timings of API calls
  - Select a timeline in the Paraver main window, click on the ‘New Histogram’ icon and select OK
- Powerful software – can also pick up your MPI communications
- Perform calculations with the data – see the Paraver user guide

Platform specific profilers

- More information can be obtained about your OpenCL program by profiling it using the hardware vendors dedicated profilers

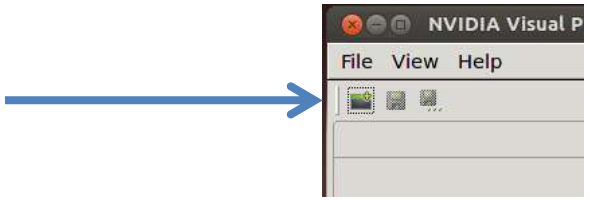
- OpenCL profiling can be done with Events in the API itself for specific profiling of queues and kernel calls

NVIDIA Visual Profiler®

This gives us information about:

- Device occupancy
- Memory bandwidth(between host and device)
- Number of registers uses
- Timeline of kernel executions and memory copies
- Etc…

- Start a new session:

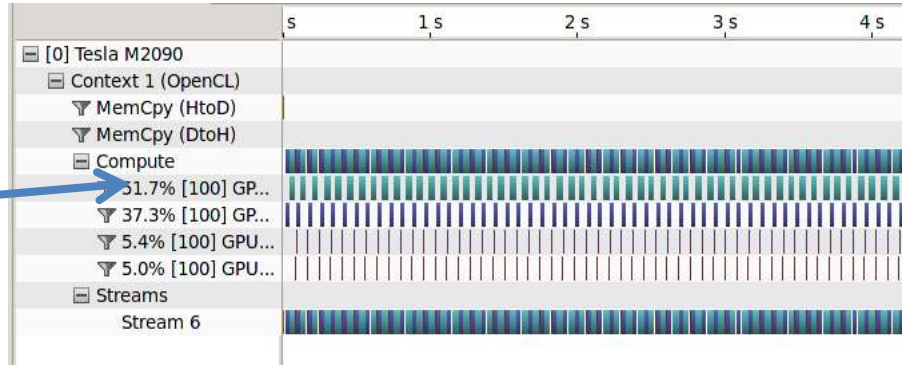


- Follow the wizard, selecting the compiled binary in the File box (you do not need to make any code or compiler modifications). You can leave the other options as the default.
- The binary is then run and profiled and the results displayed.

Profiling using nvvp

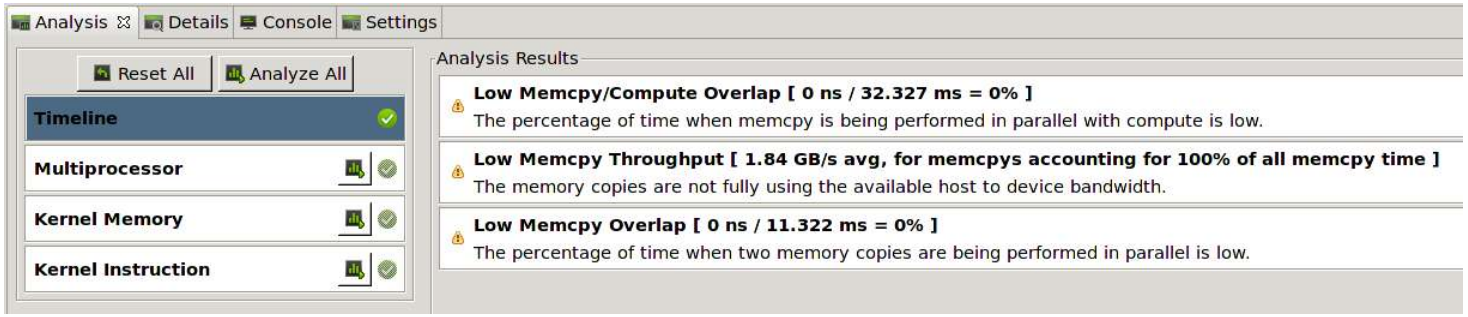
- The timeline says what happened during the program execution:

Kernels



Each invocation of the kernel is pictured as a box

- Some things to think about optimising are displayed in the Analysis tab:



Profiling using nvvp

- The Details tab shows information for each kernel invocation and memory copy
  - number of registers used
  - work group sizes
  - memory throughput
  - amount of memory transferred

- No information about which parts of the kernel are running slowly, but the figures here might give us a clue as to where to look

- Best way to learn: experiment with an application yourself

Profiling from the command line

- NVIDIA® also have `nvprof` and 'Command Line Profiler'
- nvprof available with CUDA™ 5.0 onwards, but currently lacks driver support for OpenCL profiling
- The legacy command-line profiler can be invoked using environment variables:

```
$ export COMPUTE_PROFILE=1
$ export COMPUTE_PROFILE_LOG=<output file>
$ export COMPUTE_PROFILE_CONFIG=<config file>
```

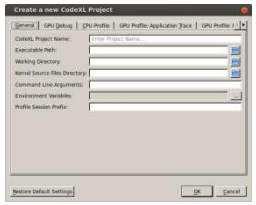
- Config file controls which events to collect (run **`nvprof --query-events`** for a comprehensive list)
- Run your application to collect event information and then inspect output file with text editor
- Can also output CSV information (**`COMPUTE_PROFILE_CSV=1`**) for inspection with a spreadsheet or import into nvvp (limited support)

AMD® CodeXL

- AMD provide a graphical Profiler and Debugger for AMD Radeon™ GPUs
- Can give information on:
  - API and kernel timings
  - Memory transfer information
  - Register use
  - Local memory use
  - Wavefront usage
  - Hints at limiting performance factors

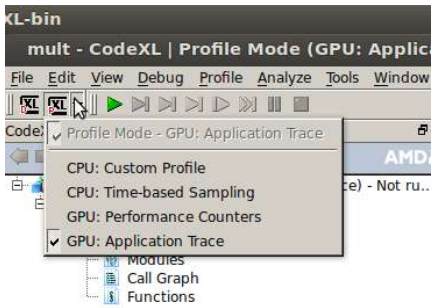
CodeXL

- Create a new project, inserting the binary location in the window



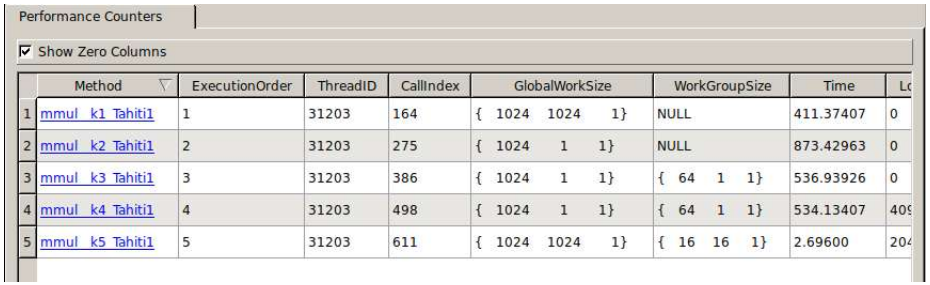
- Click on the Profiling button, and hit the green arrow to run your program

- Select the different traces to view associated information



CodeXL

- GPU: Performance Counters
  - Information on kernels including work group sizes, registers, etc.
  - View the kernel instruction code
    - Click on the kernel name in the left most column
  - View some graphs and hints about the kernel
    - Click on the Occupancy result

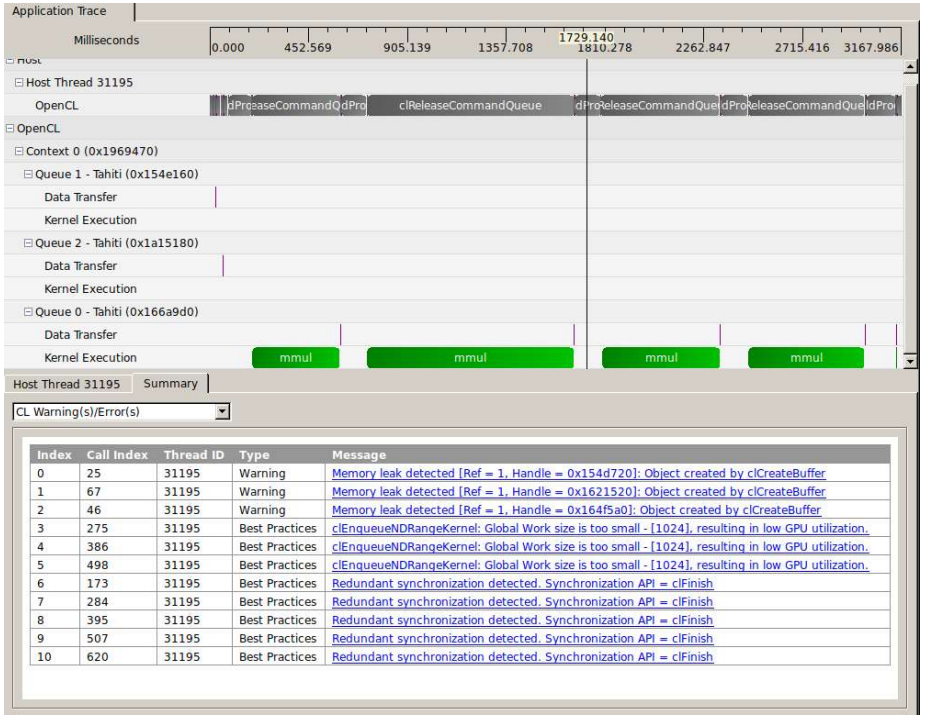


	Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	Lc
1	mmul_k1_Tahiti1	1	31203	164	{ 1024 1024 1}	NULL	411.37407	0
2	mmul_k2_Tahiti1	2	31203	275	{ 1024 1 1}	NULL	873.42963	0
3	mmul_k3_Tahiti1	3	31203	386	{ 1024 1 1}	{ 64 1 1}	536.93926	0
4	mmul_k4_Tahiti1	4	31203	498	{ 1024 1 1}	{ 64 1 1}	534.13407	409
5	mmul_k5_Tahiti1	5	31203	611	{ 1024 1024 1}	{ 16 16 1}	2.69600	204

Variable	Value	Device Limit
Device Info		
Device name	Tahiti	
Number of compute units	32	
Max number of waves per compute unit	40	
Max number of work-groups per compute unit	16	
Wavefront size	64	
Kernel Info		
Kernel name	mmul	
Vector GPR usage per work-item	31	256
Scalar GPR usage per work-item	28	102
LDS usage per work-group	2048	65536
Flattened work-group size	256	256
Flattened global work size	1048576	16777216
Number of waves per work-group	4	4
Kernel Occupancy		
Number of waves limited by Vector GPR and Work-group size	32	40
Number of waves limited by Scalar GPR and Work-group size	40	40
Number of waves limited by LDS and Work-group size	40	40
Number of waves limited by Work-group size	40	40
Limiting factor(s)	**VGPR**	
Estimated occupancy	80%	

CodeXL

- GPU: Application Trace
  - See timing information about API calls
  - Timings of memory movements
  - Timings of kernel executions



Exercise 12: Profiling OpenCL programs

- Goal:
  - To experiment with profiling tools
- Procedure:
  - Take one of your OpenCL programs, such as matrix multiply
  - Run the program in the profiler and explore the results
  - Modify the program to change the performance in some way and observe the effect with the profiler
  - Repeat with other programs if you have time
- Expected output:
  - Timings reported by the host code and via the profiling interfaces should roughly match