CS 6023 - GPU Programming

Debugging and Profiling

07/11/2018

Setting and Agenda

- What is difficult in debugging CUDA code
  - How to use cuda-gdb and cuda-memcheck
- How to profile CUDA code
  - How to use Nvidia’s nvprof

Debugging parallel code

- Debugging parallel code is hard
- Why?

Debugging parallel code

- Debugging parallel code is hard
- Why?
  - Split between device and host
  - Multiple threads running in parallel
    - Several PCs -> how to breakpoint, step, etc.
    - Thread execution order is uncertain
    - Partial results can vary -> how to use assert

Debugging parallel code

- Debugging parallel code is hard
- Why?
  - Split between device and host
  - Multiple threads running in parallel
    - Several PCs -> how to breakpoint, step, etc.
    - Thread execution order is uncertain
    - Partial results can vary -> how to use assert
- What can we do without a debugger?

Debugging parallel code

- Debugging parallel code is hard
- Why?
  - Split between device and host
  - Multiple threads running in parallel
    - Several PCs -> how to breakpoint, step, etc.
    - Thread execution order is uncertain
    - Partial results can vary -> how to use assert
- What can we do without a debugger?
  - printf on device (allowed since compute >= 4)
  - Check return values of CUDA API calls

Debugging on GPUs

- Tools
  - CUDA GDB
  - CUDA Memcheck
  - Parallel NSight
- IDE support
  - Emacs
  - DDD (Data display debugger)

CUDA GDB

- Extends standard GDB
  - Works with CUDA C/C++ and CUDA Fortran
  - Works on Linux/Mac
- Features
  - Single session to debug CPU and GPU code
  - All features of standard GDB are available
  - Breakpoints, stepping, assertions, memory inspection, coredump, error detection
  - Support multiple GPUs, multiple kernels

For machines with single card

- If the card is powering the display, can lead to screen freezes
- Best to disable X11 server
  - Run in console mode
  - Or debug remotely through ssh
- Or use a machine with multiple GPUs
  - Our lab setup is fine - as we are using a separate card for the display

Starting CUDA GDB

- Compile code with debug flags
  - -g for host code
  - -G for device code
  - `$ nvcc -g -G myProg.cu -o myProg`
- This compiles the program without any optimizations with the exception of dead-code eliminations and register-spilling optimizations
- Run the debugger
  - `$ cuda-gdb myProg`
  - `(cuda-gdb)`

Managing the execution

- Similar to how gdb works
- Launch the application
  - `$ (cuda-gdb) run`
- Continue running all threads on device and host
  - `$ (cuda-gdb) continue`
- Kill application
  - `$ (cuda-gdb) kill`
- At any time CTRL-C interrupts the debugger

Stepping through code

- We would like to step through code to debug. But with multiple threads/warps/blocks how do we do this?

Stepping through code

- We would like to step through code to debug. But with multiple threads/warps/blocks how do we do this?
  - Cuda GDB maintains **focus** of a single thread
  - How do we identify a thread?

Stepping through code

- We would like to step through code to debug. But with multiple threads/warps/blocks how do we do this?
  - Cuda GDB maintains **focus** of a single thread
  - How do we identify a thread?
    - Thread given by hardware coordinates - lane, warp, SM, device
    - Thread given by software coordinates - thread, block, grid, kernel

Seeing and setting focus

```
(cuda-gdb) cuda device sm warp lane block thread
block (0,0,0), thread (0,0,0), device 0, sm 0, warp 0, lane 0
(cuda-gdb) cuda kernel block thread
kernel 1, block (0,0,0), thread (0,0,0)
(cuda-gdb) cuda kernel
kernel 1
```

```
(cuda-gdb) cuda device 0 sm 1 warp 2 lane 3
[Switching focus to CUDA kernel 1, grid 2, block (8,0,0), thread
(67,0,0), device 0, sm 1, warp 2, lane 3]
374 int totalThreads = gridDim.x * blockDim.x;
(cuda-gdb) cuda block 1 thread 3
[Switching focus to CUDA kernel 1, grid 2, block (1,0,0), thread (3,0,0),
device 0, sm 3, warp 0, lane 3]
374 int totalThreads = gridDim.x * blockDim.
```

Stepping

- Step into function calls of all active threads in the warp of current focus
  - `$ (cuda-gdb) step`
- Step over function calls of all active threads in the warp of current focus
  - `$ (cuda-gdb) next`

Stepping

- Step into function calls of all active threads in the warp of current focus
  - `$ (cuda-gdb) step`
- Step over function calls of all active threads in the warp of current focus
  - `$ (cuda-gdb) next`
- What happens when we step at a barrier, i.e. `__syncthreads()`
  - Applies to all warps in the block of current focus
- Auto-step
  - `$ (cuda-gdb) autostep myProg.cu:25 for 20 lines`

Breakpoints

- Symbolically, i.e., at the start of functions
  - `$ (cuda-gdb) break myFunc`
- At a specific line of code
  - `$ (cuda-gdb) break myProc.cu:12`
- At a specific address
  - `$ (cuda-gdb) break *0x1afe34d0`
- At all kernel entries
  - `$ (cuda-gdb) set cuda break_on_launch application`

Conditional breakpoints

- Can make it conditional for the debugger to report breakpoint hits
  - `(cuda-gdb) break myFunc if ThreadId.x==5`
  - `(cuda-gdb) break myProg.cu:13 if i>2 && BlockId.x==0`
- Can refer to any variable including CUDA built-in variables
  - Useful for breaking only at specific points - eg. last iteration of loop

Inspecting state

- Current devices with the one in focus (*)

```
(cuda-gdb) info cuda devices
Dev  Desc  Type SMs Wps/SM Lns/Wp Regs/Ln Active SMs  Mask
* 0 gf100 sm_20  14     48     32      64            0xfff
  1 gt200 sm_13  30     32     32     128              0x0
```

- Current kernels with the one in focus (*)

```
(cuda-gdb) info cuda kernels
Kernel Dev Grid    SMs      Mask  GridDim  BlockDim  Name Args
*    1   0    2          0x3fff (240,1,1) (128,1,1) acos ...
     2   0    3          0x4000 (240,1,1) (128,1,1) asin ...
```

Inspecting state

- Current threads in a specific kernel with the one in focus (*)

```
(cuda-gdb) info cuda threads kernel 2
    Block Thread To  Block Thread Cnt        PC Filename Line
* (0,0,0) (0,0,0)  (3,0,0) (7,0,0) 32 0x7fae70  acos.cu  380
  (4,0,0) (0,0,0)  (7,0,0) (7,0,0) 32 0x7fae60  acos.cu  377
```

- Similarly for SMs, warps, lanes, etc.

Inspect stack of calls

- Stack of all calls

```
(cuda-gdb) info stack
#0 fibo_aux (n=6) at fibo.cu:88
#1 0x7bbda0 in fibo_aux (n=7) at fibo.cu:90
#2 0x7bbda0 in fibo_aux (n=8) at fibo.cu:90
#3 0x7bbda0 in fibo_aux (n=9) at fibo.cu:90
#4 0x7bbda0 in fibo_aux (n=10) at fibo.cu:90
#5 0x7cfdb8 in fibo_main<<<(1,1,1),(1,1,1)>>> (...) at fibo.cu:95
```

Inspecting memory

- Variables can be stored in register or local, shared, const or global memory
- All of these can be accessed and set
- Reading a variable

```
(cuda-gdb) print myVar
$1 = 3
(cuda-gdb) print &myVar
$2 = (@global int *) 0x200200020
(cuda-gdb) print array[3] @ 4
# 4 consecutive values of the array
```

- Setting a variable

```
(cuda-gdb) print myVar = 5
$3 = 5
```

Cuda-memcheck

- Cuda-memcheck
  - Similar in intention to `valgrind`
  - Run-time memory error checker
  - `(cuda-gdb) set cuda memcheck on`

Cuda-memcheck

- Cuda-memcheck
  - Similar in intention to `valgrind`
  - Run-time memory error checker
  - `(cuda-gdb) set cuda memcheck on`
  - Catches variety of runtime errors including
    - Stack overflow, illegal addresses (global/shared), illegal PC, …
  - Some of the errors are not precise (no threadid and incorrect PC)
    - Better to step through the code where some error is expected

Example memcheck error

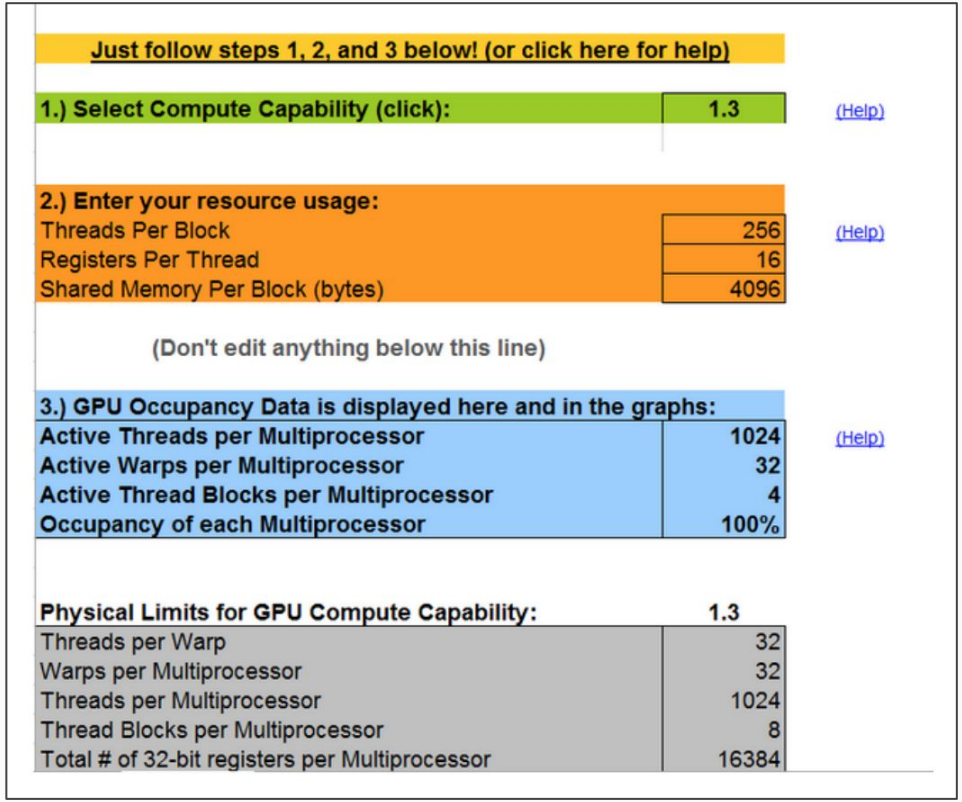
```
(cuda-gdb) set cuda memcheck on

(cuda-gdb) run
[Launch of CUDA Kernel 0 (applyStencil1D) on Device 0]
Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.
applyStencil1D<<<(32768,1,1),(512,1,1)>>> at stencil1d.cu:60

(cuda-gdb) info line stencil1d.cu:60
out[ i ] += weights[ j + RADIUS ] * in[ i + j ];
```

(Dated) Occupancy calculator

- Simple Excel file to compute occupancy of different hardware units
- Available at link
- Studies the tradeoffs across
  - Threads
  - Shared memory
  - Global memory
  - Constant memory
  - Registers
  - Blocks



Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1.3	(Help)

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	16	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	1.3
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384

Profiling

- Command-line tool nvprof
- Visual profilers: NSight, NVVP
- Profiling does not require any additional instrumentation
  - CUDA runtime by default maintains logs on several events (~140)

nvprof

- Execution
  - `$ nvcc -g -G myProg.cu -o myProg`
  - `$ nvprof myProg`
- By default profiles the entire program
  - Can be tedious if your code is small part of a larger code
- To profile part of the code “focussed profiling”
  - `#include cuda_profiler_api.h`
  - `cudaProfilerStart()` and `cudaProfilerStop()`
  - `nvprof --profile-from-start off myProg`

nvprof

- Profiler output gives you the time taken by different kernels and memcopies

```
==9261== Profiling application: ./tHogbomCleanHemi
==9261== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 58.73%  737.97ms      1000  737.97us  424.77us  1.1405ms  subtractPSFLoop_kernel(float const *, int, float*, int,
 38.39%  482.31ms      1001  481.83us  475.74us  492.16us  findPeakLoop_kernel(MaxCandidate*, float const *, int)
  1.87%  23.450ms         2  11.725ms  11.721ms  11.728ms  [CUDA memcpy HtoD]
  1.01%  12.715ms      1002  12.689us  2.1760us  10.502ms  [CUDA memcpy DtoH]
```

https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

Sample use-case

- `nvprof --metrics gld_throughput,gld_efficiency,achieved_occupancy ./sumMatrix dimX dimY`
- Running it for different dimX and dimY gives different execution times

```
⚙ cfchen@cfchen-MBP ~/Downloads/CodeSamples/chapter03 ./sumMatrix 4 4
sumMatrixOnGPU2D <<<(1024,1024), (4,4)>>> elapsed 5.806647 s
⚙ cfchen@cfchen-MBP ~/Downloads/CodeSamples/chapter03 ./sumMatrix 4 8
sumMatrixOnGPU2D <<<(1024,512), (4,8)>>> elapsed 3.085108 s
⚙ cfchen@cfchen-MBP ~/Downloads/CodeSamples/chapter03 ./sumMatrix 8 4
sumMatrixOnGPU2D <<<(512,1024), (8,4)>>> elapsed 2.559193 s
⚙ cfchen@cfchen-MBP ~/Downloads/CodeSamples/chapter03 ./sumMatrix 8 8
sumMatrixOnGPU2D <<<(512,512), (8,8)>>> elapsed 1.624973 s
⚙ cfchen@cfchen-MBP ~/Downloads/CodeSamples/chapter03 ./sumMatrix 16 16
sumMatrixOnGPU2D <<<(256,256), (16,16)>>> elapsed 1.234786 s
⚙ cfchen@cfchen-MBP ~/Downloads/CodeSamples/chapter03 ./sumMatrix 32 32
sumMatrixOnGPU2D <<<(128,128), (32,32)>>> elapsed 1.382378 s
```

https://www.ee.columbia.edu/~cylin/course/bigdata/EECS6895-AdvancedBigDataAnalytics-Lecture9.pdf

Sample use-case

- Metrics reveal the real causes

```
sumMatrixOnGPU2D <<<(1024,1024), (4,4)>>> elapsed 498.796492 ms
==78074== Profiling application: ./sumMatrix 4 4
==78074== Profiling result:
==78074== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GT 650M (0)"
       Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)
       100                            gld_throughput                   Global Load Throughput  2.5676GB/s  2.5912GB/s  2.5883GB/s
       100                            gld_efficiency             Global Memory Load Efficiency      50.00%      50.00%      50.00%
       100                         achieved_occupancy                    Achieved Occupancy    0.224382    0.224437    0.224409
```

```
sumMatrixOnGPU2D <<<(512,512), (8,8)>>> elapsed 209.860447 ms
==78025== Profiling application: ./sumMatrix 8 8
==78025== Profiling result:
==78025== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GT 650M (0)"
       Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)
       100                            gld_throughput                   Global Load Throughput  3.4194GB/s  8.1859GB/s  3.9721GB/s
       100                            gld_efficiency             Global Memory Load Efficiency      100.00%     100.00%     100.00%
       100                         achieved_occupancy                    Achieved Occupancy    0.446429    0.451606    0.447626
```

https://www.ee.columbia.edu/~cylin/course/bigdata/EECS6895-AdvancedBigDataAnalytics-Lecture9.pdf

Sample use-case

- Metrics reveal the real causes

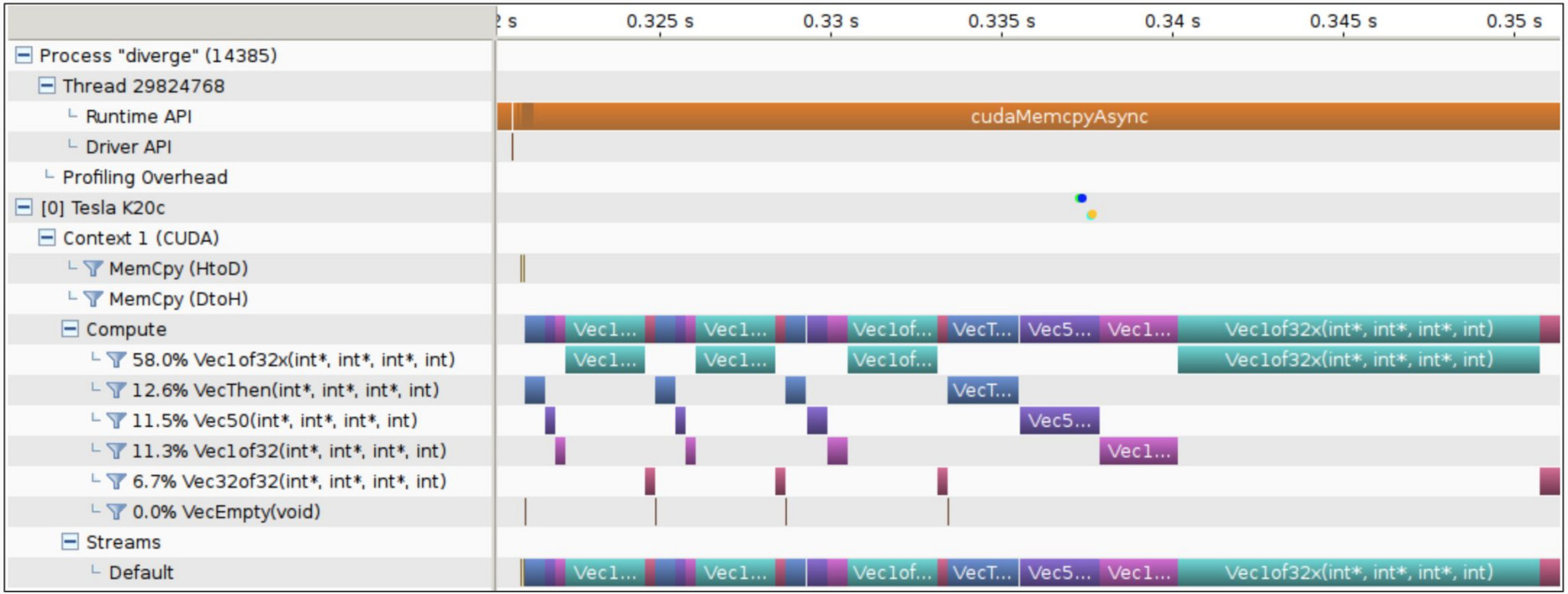
```
sumMatrixOnGPU2D <<<(256,256), (16,16)>>> elapsed 154.965583 ms
==77956== Profiling application: ./sumMatrix 16 16
==77956== Profiling result:
==77956== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GT 650M (0)"
        Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)
        100                            gld_throughput                    Global Load Throughput  4.8073GB/s  5.0724GB/s  4.9692GB/s
        100                            gld_efficiency             Global Memory Load Efficiency      100.00%     100.00%     100.00%
        100                         achieved_occupancy                     Achieved Occupancy    0.821609    0.847440    0.845807
```

```
sumMatrixOnGPU2D <<<(128,128), (32,32)>>> elapsed 182.770676 ms
==77762== Profiling application: ./sumMatrix 32 32
==77762== Profiling result:
==77762== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "GeForce GT 650M (0)"
        Kernel: sumMatrixOnGPU2D(float*, float*, float*, int, int)
        100                            gld_throughput                    Global Load Throughput  4.6957GB/s  5.0193GB/s  4.8495GB/s
        100                            gld_efficiency             Global Memory Load Efficiency      100.00%     100.00%     100.00%
        100                         achieved_occupancy                     Achieved Occupancy    0.753967    0.776110    0.772071
```

https://www.ee.columbia.edu/~cylin/course/bigdata/EECS6895-AdvancedBigDataAnalytics-Lecture9.pdf

Visual profiler

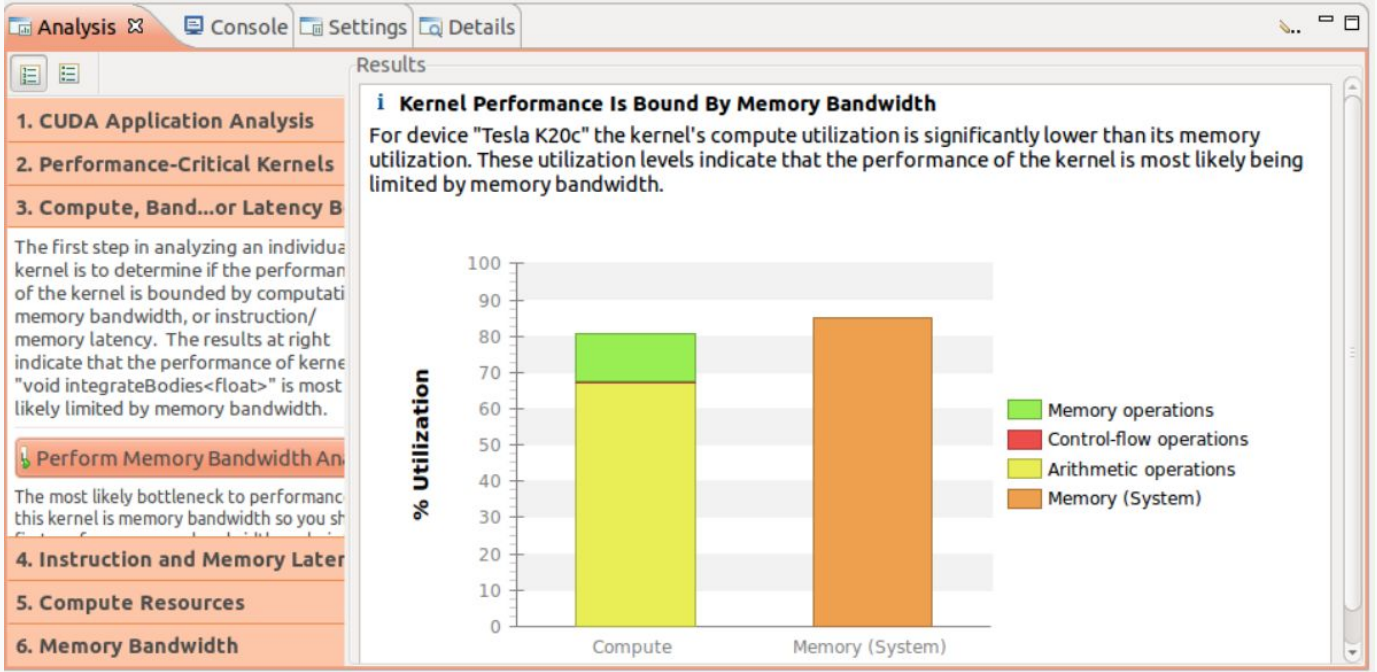
- Launch nvpp and import a session (output file) from `nvprof`



https://docs.nvidia.com/cuda/profiler-users-guide/index.html

Analysis view

- `nvprof --analysis-metrics -o nbody-analysis.nvprof ./nbody --benchmark -numdevices=2 -i=1`

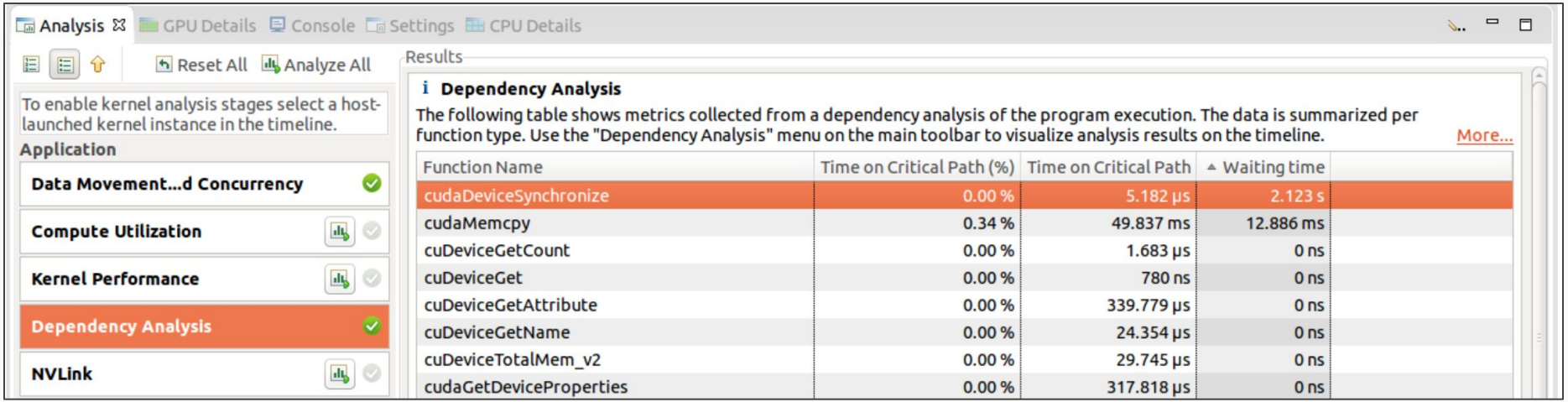


https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

Visualizing concurrency with streams

Dependency analysis

- Create a dependency graph of the application
  - Eg. synchronous kernel invocation, barrier sync
- Helps identify critical path, contributors to it, and waiting times



A quick recipe for performance optimization - for projects

- Must-do
  - Parallelise sequential code (remember Amdahl’s law)
  - Memory bandwidth between CPU and GPU
    - Reduce amount transferred
    - Hide latency with asynchronous transfers
  - Use shared memory effectively
    - Privatisation
    - Tiling
  - If required to use global memory, make coalesced and aligned accesses
  - Reorder work by thread to reduce control divergence within warps

A quick recipe for performance optimization - for projects

- Good to do (applies mostly to serial code too)
  - Use libraries to speed up operations
  - Simple computation
    - Use intrinsic functions when available (note: loss of accuracy)
      - `__sinf()` instead of `sinf()`
    - Use lower precision arithmetic, eg. 32 instead of 64 bit
    - Use bitwise operations and rotations
  - Loop unrolling
  - ...