CUDA Tools

Tools for Profiling and Debugging

Nikolaos Nikoloutsakos

GRNET

Athens, 14 March 2019

Contents

Introduction

- Improve what you can measure, need to identify:
  - hotspots: which function takes most of the run item.
  - bottlenecks: what limits the performance of the hotspots.
- Manual timing/debugging is difficult
- Access to hardware counters/events

CUDA tools:

- cuda-memcheck to detect invalid memory access
- cuda-gdb, nsight eclipse to debug a CUDA program.
- nvprof, visual profiler to profile CUDA programs
- occupancy calculator

CUDA Debugging Tools

cuda-memcheck

- Detects memory access errors
- Run time error detection
- Included in CUDA Toolkit
- Similar to Valgrind
- `http://docs.nvidia.com/cuda/cuda-memcheck/`

cuda-memcheck

Errors

- Memory access
  - out of bound or misaligned access to memory
  - shared memory hazard checking
- Hardware exception errors
- Malloc/Free errors
  - due to incorrect use of malloc or free
  - allocations of device memory which have not been freed
- Leak errors
  - device heap memory leaks
  - cudaMalloc allocation leak
- CUDA API errors
  - failure of cuda API call

cuda-memcheck

Getting started:

- compile with `-g` (host debug info) `-G` (device)
  - Line number information
  - Full debug information (variables,functions etc)
  - Disables Optimizations
- `cuda-memchek a.out`

```
========= CUDA-MEMCHECK
Mallocing memory
Running unaligned_kernel
Ran unaligned_kernel: no error
========= Invalid __global__ write of size 4
=========     at 0x00000038 in memcheck_demo.cu:6:unaligned_kernel(void)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x0d260001 is misaligned
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame:/usr/lib/libcuda.so (cuLaunchKernel + 0x331) [0x138291]
```

cuda-memcheck

- `__global__`: for device global memory
  `__shared__`: for per block shared memory
  `__local__`: for per thread local memory
- the source file and line number
- information about type of access `(read / write)`
- the thread indices and block indices of the thread
- memory address being accessed and the type of of access error

More options

- leak checking with: `cuda-memcheck -leak-check full`
  - allocation not freed at `cuCtxDestroy`
- api(runtime/driver) error checking:
  `cuda-memcheck -report-api-errors yes`
  - enabled by default,
- racecheck analysis mode: `cuda-memcheck -tool racecheck -racecheck-report analysis`
  - sharing data between threads.

cuda-gdb I

- Debug CUDA appplications running on actual hardware
- extension of GNU gdb
- `http://docs.nvidia.com/cuda/cuda-gdb/`
- Gui Tools: Nvidia nsight, or DDD (`ddd -debugger cuda-gdb`)

Getting started:

- `nvcc -g -G foo.cu -o foo`
  - forces `-O0 compilation`
  - includes debug information
- Start the execution control: `cuda-gdb foo`

cuda-gdb II

- `(cuda-gdb) run`: starts application
- `(cuda-gdb) break my_kernel` : break execution at function name

```
(cuda—gdb) break vectorAdd
Breakpoint 1 at 0x402cfe
```

- `(cuda-gdb) continue`: resume the application
- `(cuda-gdb) kill` : kill the application (interrupt 'CTRL-C')
- `(cuda-gdb) info locals`: print all currently set variables
- `(cuda-gdb) info cuda devices`: obtain list of devices in the system

```
  Dev Description SM Type SMs Warps/SM Lanes/Warp Max Regs/Lane Active SMs Mask
*   0       GF110   sm_20  14       48         32            64 0x00003fff^^I^^I
```

cuda-gdb III

- `(cuda-gdb) info cuda kernels`: all active kernels

```
  Kernel  Parent  Dev  Grid  Status    SMs Mask    GridDim    BlockDim  Invocation
*      0       —    0     1  Active  0x00003fff  (196,1,1)  (256,1,1)  vectorAdd()
```

- `(cuda-gdb) info cuda threads`: active CUDA blocks and threads with the total count of threads in those blocks

```
  BlockIdx  ThreadIdx  To  BlockIdx  ThreadIdx  Count          Virtual PC  Filename  Line
Kernel 0
*  (0,0,0)    (0,0,0)      (83,0,0)  (255,0,0)  21504  0x0000000000818a50       n/a     0
```

- `(cuda-gdb) cuda kernel 0 block 1,0,0 thread 3,0,0`: switch focus to any currently running thread

cuda-gdb IV

```
[Switching focus to CUDA kernel 0, grid 1, block (1,0,0), thread (3,0,0), device 0, sm
     5, warp 0, lane 3]
0x0000000000818a50 in vectorAdd(float const*, float const*, float*, int)<<<(196,1,1)
     ,(256,1,1)>>> ()
```

- `(cuda-gdb) cuda kernel block thread`: obtain the current focus:

```
kernel 0, block (1,0,0), thread (3,0,0)
```

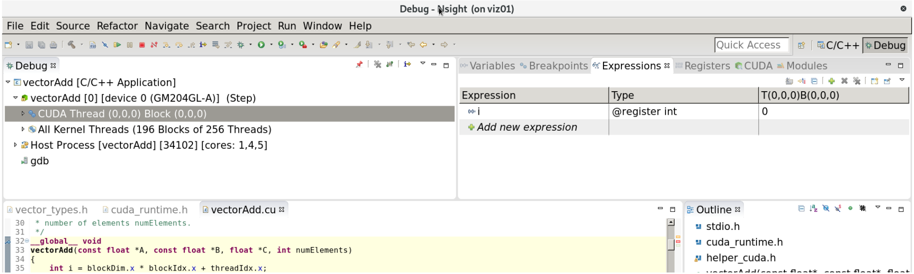
- `(cuda-gdb) info stack`: stack trace

```
#0  0x0000000000818a50 in vectorAdd(float const*, float const*, float*, int)<<<(196,1,1)
     ,(256,1,1)>>> ()
```

- `(cuda-gdb) print **variable**`: source variables

Nsight Eclipse Edition

- full-featured IDE powered by the Eclipse platform
- build system
- gui for debugging (using the cuda-gdb)
- https://developer.nvidia.com/nsight-eclipse-edition



CUDA Profiling Tools

Profiling

- is the app running kernels on the GPU at all?
- is it performing excessive memory copies?
- identify:

Hotspot

An area of code within the program that uses a disproportionately high amount of processor time.

Bottleneck

An area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays.

CUDA Profiling Tools

NVIDIA Visual Profiler (nvvp)

is a graphical profiling tool that displays a timeline of your application's CPU and GPU activity, and that includes an automated analysis engine to identify optimization opportunities.

Command-line profiling tool (nvprof)

The `nvprof` profiling tool enables you to collect and view profiling data from the command-line.

Timing by Hand

CPU, GPU Timers

Timing by Hand

- Using CPU timers (note, some CUDA functions are asynchronous `cudaDeviceSyncronize()`)
- CUDA GPU Timers

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord( start, 0 );
kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y,
NUM_REPS);
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

nvprof

Getting started:

- `nvprof ./a.out`

```
==61089== Profiling application: ./matrixMul
==61089== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.78%  86.906ms       301  288.73us  286.07us  292.03us  void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
  0.13%  114.14us         2  57.070us  39.199us  74.942us  [CUDA memcpy HtoD]
  0.09%  74.623us         1  74.623us  74.623us  74.623us  [CUDA memcpy DtoH]

==61089== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 78.90%  341.90ms         3  113.97ms  264.25us  341.36ms  cudaMalloc
 19.00%  82.319ms         1  82.319ms  82.319ms  82.319ms  cudaEventSynchronize
  0.82%  3.5671ms       301  11.850us  10.844us  111.75us  cudaLaunch
  0.35%  1.4999ms       182  8.2410us     197ns  351.92us  cuDeviceGetAttribute
  0.25%  1.0689ms         3  356.29us  148.57us  717.16us  cudaMemcpy
  0.17%  742.11us         1  742.11us  742.11us  742.11us  cudaGetDeviceProperties
  0.16%  709.69us      1505     471ns     393ns  20.747us  cudaSetupArgument
  0.13%  544.07us         3  181.36us  174.60us  193.24us  cudaFree
  0.09%  401.32us         2  200.66us  193.96us  207.35us  cuDeviceTotalMem
  0.06%  271.72us         1  271.72us  271.72us  271.72us  cudaDeviceSynchronize
  0.03%  140.01us       301     465ns     413ns  2.5240us  cudaConfigureCall
  0.02%  107.05us         2  53.526us  49.401us  57.651us  cuDeviceGetName
  0.01%  29.004us         1  29.004us  29.004us  29.004us  cudaGetDevice
  0.01%  23.916us         2  11.958us  1.9860us  21.930us  cudaEventCreate
```

nvprof

- *Summary Mode* (default): single result line for each kernel function and each type of CUDA memory copy/set performed by the application.
- *Trace* `-print-gpu-trace`: . GPU-Trace mode provides a timeline of all activities taking place on the GPU in chronological order. see on which GPU each kernel ran, as well as the grid dimensions used for each launch.

```
==66063== Profiling application: ./nbody --benchmark -numdevices=2 -i=1
==66063== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device   Context    Stream  Name
691.48ms     832ns                    -               -         -         -         -        4B  4.5850MB/s  Quadro M5000 (0         2        15  [CUDA memcpy HtoD]
691.50ms  1.2480us                    -               -         -         -         -        4B  3.0566MB/s  Quadro M5000 (1         1         7  [CUDA memcpy HtoD]
691.53ms     832ns                    -               -         -         -         -        4B  4.5850MB/s  Quadro M5000 (0         2        15  [CUDA memcpy HtoD]
691.54ms     800ns                    -               -         -         -         -        4B  4.7684MB/s  Quadro M5000 (1         1         7  [CUDA memcpy HtoD]
708.95ms  1.4720us                    -               -         -         -         -        8B  5.1830MB/s  Quadro M5000 (0         2        15  [CUDA memcpy HtoD]
708.96ms  1.2480us                    -               -         -         -         -        8B  6.1133MB/s  Quadro M5000 (1         1         7  [CUDA memcpy HtoD]
708.99ms     832ns                    -               -         -         -         -        8B  9.1699MB/s  Quadro M5000 (0         2        15  [CUDA memcpy HtoD]
709.00ms     832ns                    -               -         -         -         -        8B  9.1699MB/s  Quadro M5000 (1         1         7  [CUDA memcpy HtoD]
723.62ms  4.8337ms             (64 1 1)       (256 1 1)        32        0B  4.0000KB         -           -  Quadro M5000 (0         2        15  void integrateBodies
<float>(vec4<float>::Type*, vec4<float>::Type*, vec4<float>::Type*, unsigned int, unsigned int, float, float, int) [277]
723.69ms  4.7640ms             (64 1 1)       (256 1 1)        32        0B  4.0000KB         -           -  Quadro M5000 (1         1         7  void integrateBodies
<float>(vec4<float>::Type*, vec4<float>::Type*, vec4<float>::Type*, unsigned int, unsigned int, float, float, int) [291]
```

nvprof

Profile anything

- `-output-profile` you can output a data file for later import into either nvprof or the NVIDIA Visual Profiler.
- `-query-events` list measure specific events to check with `-events`
- `-query-metrics` list available metrics to check with `-metrics`
- `-analysis-metrics` capture all of the GPU metrics that the Visual Profiler needs for its ”guided analysis” mode.
- `-help` help information.
- `http://docs.nvidia.com/cuda/profiler-users-guide/`

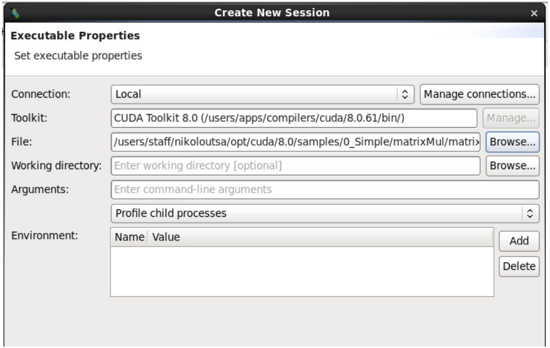
nvvp

- Visualize and optimize performance of a CUDA application
- timeline view on CPU and GPU
- import data generated by nvprof
- guided and unguided analysis
- `https://developer.nvidia.com/nvidia-visual-profiler`

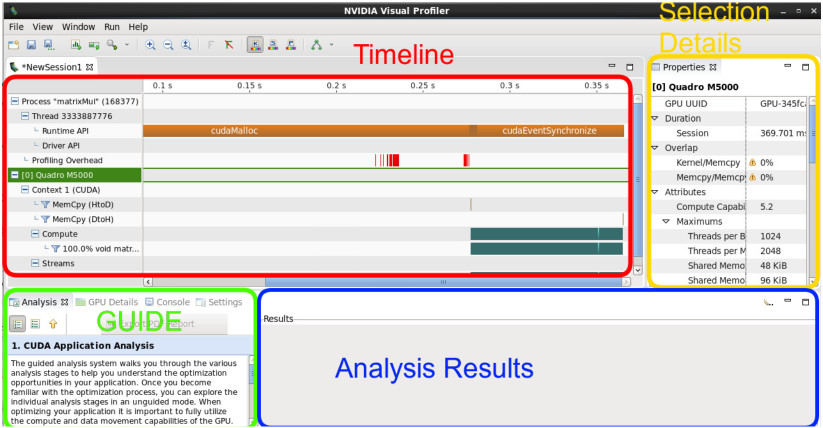
nvvp

Getting started:

- `nvvp` -> File -> New Session

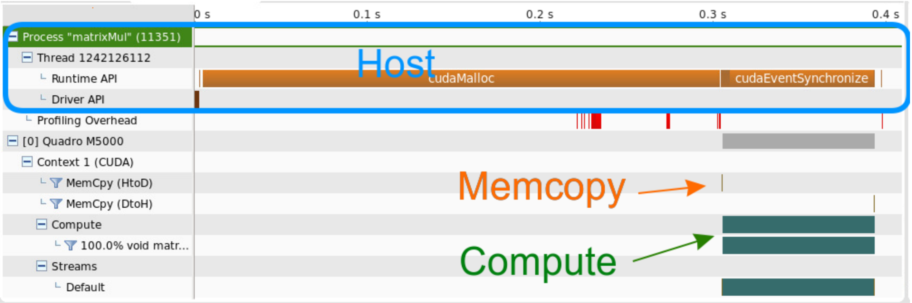


Overview



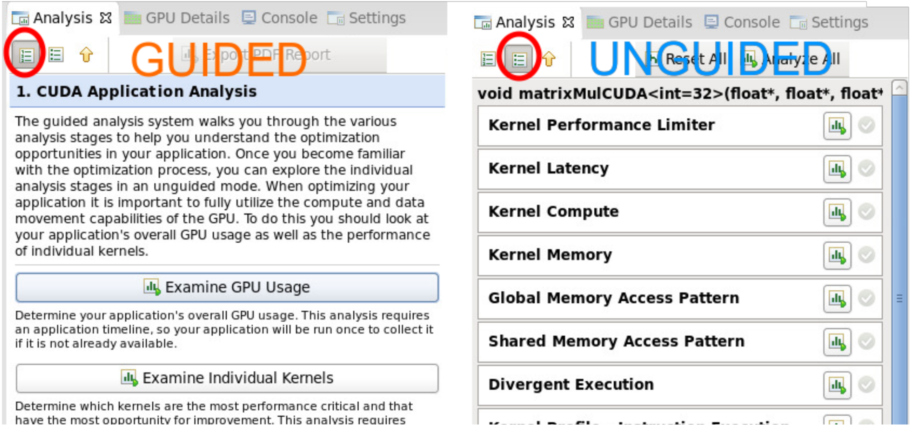
NVIDIA Visual Profiler
File View Window Run Help
Timeline
Selection Details
*NewSession1
Properties
[0] Quadro M5000
0.1 s
0.15 s
0.2 s
0.25 s
0.3 s
0.35 s
Process "matrixMul" (168377)
Thread 3333887776
Runtime API
cudaMalloc
cudaEventSynchronize
Driver API
Profiling Overhead
[0] Quadro M5000
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
100.0% void matr...
Streams
GPU UUID
GPU-345fc
Duration
Session
369.701 m
Overlap
Kernel/Memcpy
0%
Memcpy/Memcp
0%
Attributes
Compute Capabi
5.2
Maximums
Threads per B
1024
Threads per M
2048
Shared Memo
48 KiB
Shared Memo
96 KiB
Analysis
GPU Details
Console
Settings
GUIDE
Export PDF Report
1. CUDA Application Analysis
The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU.
Results
Analysis Results

Timeline

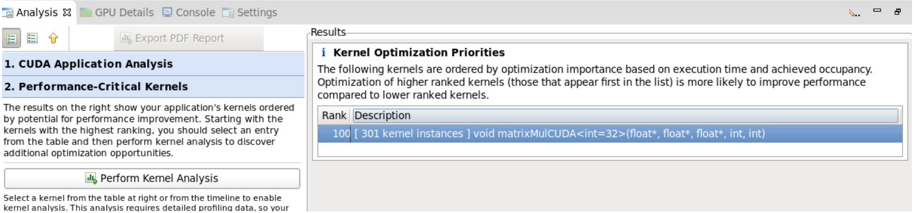


0 s
0.1 s
0.2 s
0.3 s
0.4 s
Process "matrixMul" (11351)
Thread 1242126112
Runtime API
Driver API
Profiling Overhead
[0] Quadro M5000
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
100.0% void matr...
Streams
Default
Host
cudaMalloc
cudaEventSynchronize
Memcopy
Compute

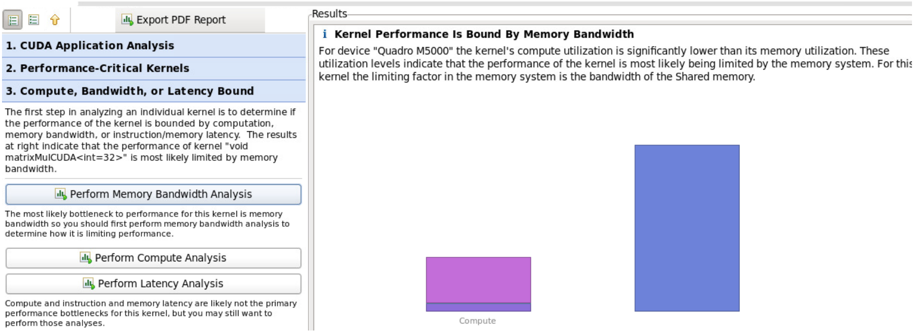
Analysis


Analysis
GPU Details
Console
Settings
GUIDED
Export PDF Report
1. CUDA Application Analysis
The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels.
Examine GPU Usage
Determine your application's overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.
Examine Individual Kernels
Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires
Analysis
GPU Details
Console
Settings
UNGUIDED
Reset All
Analyze All
void matrixMulCUDA<int=32>(float*, float*, float*
Kernel Performance Limiter
Kernel Latency
Kernel Compute
Kernel Memory
Global Memory Access Pattern
Shared Memory Access Pattern
Divergent Execution

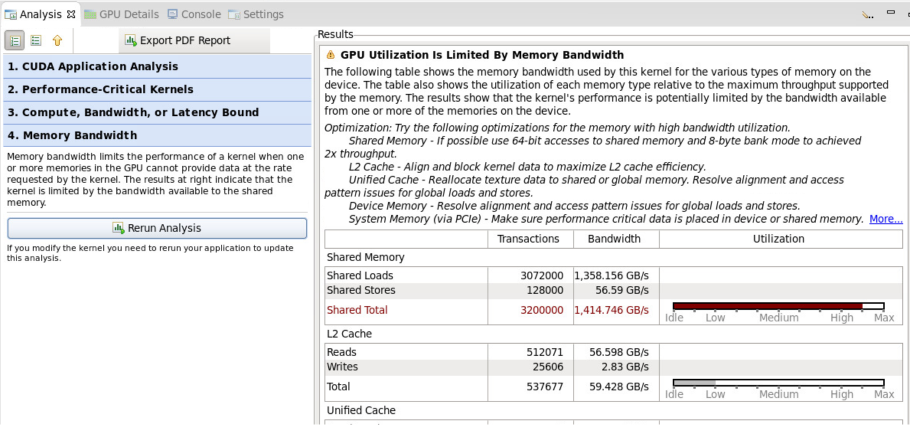
Examine Kernels



Utilisation



Memory Bandwidth analysis



Help

Help | Contents | Search | Related Topics | Bookmarks | Index

2.4.4. GPU Details View

The GPU Details View displays a table of information for each memory copy and kernel execution in the profiled application. The following figure shows the table containing several memcpy and kernel executions. Each row of the table contains general information for a kernel execution or memory copy. For kernels, the table will also contain a column for each metric or event value collected for that kernel. In the figure, the **Achieved Occupancy** column shows the value of that metric for each of the kernel executions.

Details

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput	Achieved Occupancy
Memcpy HtoD [async]	518.069 ms	46.528 µs	n/a	n/a	n/a	n/a	n/a	256 KB	5.25 GB/s	n/a
Memcpy HtoD [async]	518.205 ms	46.367 µs	n/a	n/a	n/a	n/a	n/a	256 KB	5.27 GB/s	n/a
VecEmpty(void)	518.704 ms	3.2 µs	[1,1,1]	[1,1,1]	6	0	0	n/a	n/a	0.016
VecThen(int*, int*, int*, int)	518.75 ms	219.295 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec50(int*, int*, int*, int)	518.971 ms	108.319 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec1of32(int*, int*, int*, int)	519.081 ms	108.095 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec1of32x(int*, int*, int*, int)	519.191 ms	1.049 ms	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016
Vec32of32(int*, int*, int*, int)	520.242 ms	108.287 µs	[1,1,1]	[1,1,1]	12	0	0	n/a	n/a	0.016

You can sort the data by column by left clicking on the column header, and you can rearrange the columns by left clicking on a column header and dragging it to its new location. If you select a row in the table, the corresponding interval will be selected in the Timeline View. Similarly, if you select a kernel or memcpy interval in the Timeline View the table will be scrolled to show the corresponding data.

If you hover the mouse over a column header, a tooltip will display the data shown in that column. For a column containing event or metric data, the

Profliling MPI+CUDA Applications

http://docs.nvidia.com/cuda/profiler-users-guide/index.html#mpi-nvprof

Collect data with `nvprof`

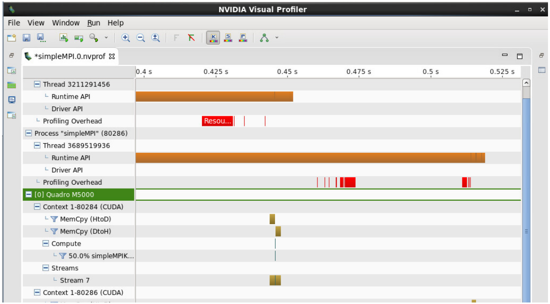
Start the `nvprof` with the mPI launcher to generate output profile for each process. string %q{ENV} can be used to name the output file
openpi: `OMPI_COMM_WORLD_RANK` , intelmpi: `PMI_RANK`, slurm: `SLURM_PROCID`

```
srun nvprof —o output.%h.%p.%q{SLURM_PROCID} <GPU_EXECUTABLE>
mpirun —np 2 nvprof —o simpleMPI.%q{PMI_RANK}.nvprof ./simpleMPI
Running on 2 nodes
==80284== NVPROF is profiling process 80284, command: ./simpleMPI
==80286== NVPROF is profiling process 80286, command: ./simpleMPI
Average of square roots is: 0.667279
PASSED
==80286== Generated result file: /users/staff/nikoloutsa/opt/cuda/8.0/samples/0_Simple/
      simpleMPI/simpleMPI.1.nvprof
==80284== Generated result file: /users/staff/nikoloutsa/opt/cuda/8.0/samples/0_Simple/
      simpleMPI/simpleMPI.0.nvprof^^I
```

Profiling MPI+CUDA Applications

Analyzing profiles with `nvvp`

Import Multi-Process nvprof Session



Occupancy

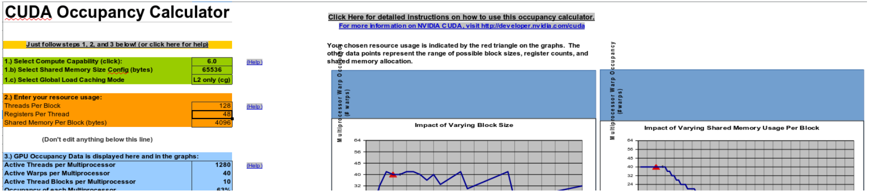
- The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU (check `deviceQuery`)
- Keeping the hardware busy helps the warp scheduler to hide latencies.
- Higher occupancy does not always equate to higher performance-there is a point above which additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.
- threads per block should be a multiple of warp size
- a minimum of 64 threads per block should be used
- 128-256 threads per block is a good starting point for further experimentation

Occupancy

Each multiprocessor on the device has a set of N registers available for use by CUDA program threads. These registers are a shared resource that are allocated among the thread blocks executing on a multiprocessor. The CUDA compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine simultaneously. If a program tries to launch a kernel for which the registers used per thread times the thread block size is greater than N, the launch will fail.

Occupancy Calculator

- /apps/compilers/cuda/8.0.61/tools/CUDA_Occupancy_Calculator.xls
- cudaOccupancyMaxPotentialBlockSize Returns grid and block size that achieves maximum potential occupancy for a device function.
- https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__OCCUPANCY.html



Thank you :)