Profiling CUDA Applications with Nvidia Nsight Systems

BY: KEVIN MAHESH KURIAKOSE

- From Nvidia: "[...] a system-wide performance analysis tool designed to visualize an application's algorithms, identify the largest opportunities to optimize, and tune to scale efficiently across [various systems]"
- Profiler/Tracer for GPU-based applications on Nvidia hardware
 - Graphics: OpenGL, OpenXR, Vulkan, DirectX
 - o Video: NVDEC, NVENC
 - o Compute: CUDA, OpenACC

What is Nsight?

- o Communication: MPI, OpenSHMEM, UCX, NCCL
- CPU: OpenMP, Python, C/C++
- Successor to NVProf

Why Profiling/Tracing?

- Shows where your program is spending its time
 - o Often, bottlenecks are in small sections of the program
 - Helps focus performance optimizations
- Tracing gives a timeline of all events
 - Very detailed, lots of data
- Profiling gives you a statistical report from sampled events
 - Useful for long-running programs

What can be profiled? (non-exhaustive)

- Kernel Executions
 - o Time Taken
 - o Grid Dimensions
 - Register/Shared Memory Usage
 - o Occupancy
- Memory Transfers
 - o Time Taken
 - Source/Dest. Type
 - o Throughput

- Communication
 - MPI/OpenSHMEM/UCX/NCCL API Calls
 - InfiniBand Transfer Metrics
- GPU Hardware Metrics
 - o GPU Context Switches
 - o GPU I/O
 - o Clock Speed
 - o Kernels in Flight
 - o Power Draw
- OS Metrics
 - o CPU Context Switches
 - CPU Instruction Pointer Sampling
 - o System Calls

How to profile

- Nsight Systems GUI
 - Can be used for profiling programs on the same machine
 - Can open profiler reports generated by the CLI
- nsys CLI utility
 - O Useful for servers/clusters with separate
 GPU nodes
 - o nsys profile [options] <program> <args...>
 - Generates an .nsys-rep file containing results
- More information in the <u>User Guide</u>

1	2	3	4	5	6	7	8	9	
	+								
10	20	30	40	50	60	70	80	90	

Example: Adding two vectors

- Out = A + B (Element-wise)
- A, B are 1GB FP32 vectors (N = 2^{28})
- Starting from Naïve implementation, profile and optimize
- Tested on an Nvidia V100 GPU

Naïve Implementation

- Direct copy of CPU code
- No GPU-specific optimizations done
- Note the kernel launch arguments:
 - o Grid size: 1
 - o Block size: 1
- Launched as nsys profile ./main

```
global___ void add_naive(...) {
  for (int i = 0; i < N; i += 1) {
    out[i] = a[i] + b[i];
  }
int main() {
  add_naive<<<1, 1>>>(
    N, dev_a, dev_b, dev_out
 );
```

Naïve Implementation

- Overall time: 15.26 seconds
 o GPU Time: 14.53 seconds
- Practically no speedup
- Reason: Only one thread used

•		4.45s 8s 12s
▶ CPU (32)	100% 0	
✓ CUDA HW (0000:62:00.0 - Tesla V	Kernel Memory	
95.7% Kernels		add_naive(int, const float *, const float *, flo
▶ 4.3% Memory		add_naive
▼ Threads (8)		Begins: 3.69605s
👻 🔽 [1211590] main 👻	0 to 100%	Ends: 18.2302s (+14.534 s) grid: <<<1, 1, 1>>> block: <<<1, 1, 1>>>
OS runtime libraries		Launch Type: Regular
CUDA API		Static Shared Memory: 0 bytes Dynamic Shared Memory: 0 bytes
Profiler overhead		Registers Per Thread: 22 Local Memory Per Thread: 0 bytes Local Memory Total: 9,43,71,840 bytes
▼ 🗸 [1211602] cuda-EvtHandlr ▼	0 to 100%	Shared Memory executed: 0 bytes Shared Memory Bank Size: 4 B
OS runtime libraries		Theoretical occupancy: 50 % Launched from thread: 1211590
	•	Latency: ←88.049 µs
Events View 👻		Correlation ID: 124
		Stream: Default stream 7

One Block Implementation

- Use one block of threads
 - o Here, 256
- Per-thread loop now jumps ahead by block size

```
_global___ void add_one_block(...) {
  int start = threadIdx.x;
  int stride = blockDim.x;
  for (int i = start; i < N; i += stride) {</pre>
    out[i] = a[i] + b[i];
  }
// ...
int main() {
  const int BLOCK_SIZE = 256;
  add_one_block<<<1, BLOCK_SIZE>>>(...);
```

One Block Implementation

- Overall time: 1.06 seconds
 - o GPU Time: 407 milliseconds
- Significant Speedup, but can be improved

► CPU (32)	100% 0	
▼ CUDA HW (0000:62:00.0 - Tesla V	Kernel Memory	
▶ 38.6% Kernels		add_one_bl
▶ 61.4% Memory		add_one_block
✓ Threads (8)		Begins: 9.15431s
✓ [1211723] main	0 to 100%	Ends: 9.56131s (+407.002 ms) grid: <<<1, 1, 1>>> block: <<<256, 1, 1>>>
OS runtime libraries		Launch Type: Regular
CUDA API	cud cud.	Static Shared Memory: 0 bytes Dynamic Shared Memory: 0 bytes
Profiler overhead		Registers Per Thread: 26 Local Memory Per Thread: 0 bytes Local Memory Total: 9,43,71,840 bytes
▼ ✓ [1211736] cuda-EvtHandlr ▼	0 to 100%	Shared Memory executed: 0 bytes Shared Memory Bank Size: 4 B
OS runtime libraries		Launched from thread: 1211723
	<u> </u>	Latency: ←92.473 µs
Events View 👻		Correlation ID: 124 Stream: Default stream 7
		oricani. Delaart stream 7

Multi Block Grid Implementation

- Use a grid of multiple blocks
 - Launch as many blocks needed to cover vectors
 - o Effectively 268,435,456 threads
- Per-thread loop now jumps ahead by grid size

```
_global___ void add_grid(...) {
  int start = (blockDim.x * blockIdx.x)
    + threadIdx.x;
  int stride = gridDim.x * blockDim.x;
  for (int i = start; i < N; i += stride) {</pre>
    out[i] = a[i] + b[i];
// ...
int main() {
  const int BLOCK_SIZE = 256;
  add_grid<<<N / BLOCK_SIZE, BLOCK_SIZE>>>(...);
```

Multi Block Grid Implementation

- Overall time: 661 milliseconds
 - o GPU Time: 4 milliseconds
- Compute is practically instant, but data transfers are slow

► CPU (32)	100% 0		
✓ CUDA HW (0000:62:00.0 - Tesla V	Kernel Memory		
▶ 0.6% Kernels		a	
▶ 99.4% Memory		1	add_grid
 Threads (8) 			Begins: 3.70489s Ends: 3.70894s (+4.050 ms)
▾ ✔ [1212552] main ▾	0 to 100%		grid: <<<1048576, 1, 1>>> block: <<<256, 1, 1>>>
OS runtime libraries			Launch Type: Regular Static Shared Memory: 0 bytes
CUDA API		cudaMemcpy . [Dynamic Shared Memory: 0 bytes
Profiler overhead		L	Registers Per Thread: 26 Local Memory Per Thread: 0 bytes Local Memory Total: 9,43,71,840 bytes
▼ ▼ [1212565] cuda-EvtHandlr ▼	0 to 100%	ç	Shared Memory executed: 0 bytes Shared Memory Bank Size: 4 B
OS runtime libraries		L L	Theoretical occupancy: 100 % Launched from thread: 1212552
Events View 👻		(Latency: ←95.178 µs Correlation ID: 124 Stream: Default stream 7

Memcpy Delay

- A, B, Out Copies: ~200 milliseconds each
 - o Throughput: ~4 GB/s
 - Vector copies are to/from Pageable memory
- Driver makes an internal copy to ensure data is guaranteed to be in memory during transfer
- Solution: Create host vectors in pinned memory (cudaMallocHost)

.		3.3	59s 3.4s	3.6s
► CPU (32)	100% 0	-		
	Kernel Memory			
▶ 0.6% Kernels				
▼ 99.4% Memory				
66.7% HtoD memcpy		Memcpy	HtoD (Pageable)	Memcpy HtoD (F
33.3% DtoH memcpy			Desine: 2.07647e	
→ Threads (8)			Begins: 3.27647s Ends: 3.49047s (+	213.996 ms)
👻 🔽 [1212552] main 👻	0 to 100%		HtoD memcpy 1,0 Source memory ki Destination memo	7,37,41,824 bytes nd: Pageable
OS runtime libraries			Throughput: 4.672	*
CUDA API				
Profiler overhead			Latency: ←274.53 Correlation ID: 122 Stream: Default st	2
	0 +- 1000	4		

Pinned Memcpy

- Overall time: 258 milliseconds
 - A, B, Out copies: ~80 milliseconds
 - Throughput: ~11 GB/s
- Many GPU applications are bottlenecked by transfers
- Further optimizations possible (ex. Multi-stream pipelining)

2s 🗸		2s 262.7ms 300m	s +400n	ns +50	
> CPU (32)	100% 0_				
- CUDA HW (0000:62:00.0 - Tesla V	Kernel Memory				
1.6% Kernels					
✓ 98.4% Memory					
68.0% HtoD memcpy		Memcpy HtoD	Memcpy HtoD		
32.0% DtoH memcpy Threads (8)		Ends: 2.317	Begins: 2.23056s Ends: 2.31706s (+86.502 ms)		
▼ 🔽 [1213227] main 👻	0 to 100%	Source men	py 1,07,37,41,824 bytes nory kind: Pinned memory kind: Device		
OS runtime libraries			:: 11.5604 GiB/s om thread: 1213227		
CUDA API		cuda Latency: ←(cudaMemcpy	
Profiler overhead		Correlation Stream: Def	ID: 125 ault stream 7		
	0 +- 100%	1			

Extras

• nsys stats <report file>

• Prints a summarized report of program statistics

• nsys analyze <report file>

• Provides suggestions for improving performance based on the report

References

- Nvidia Nsight Systems Homepage: <u>https://developer.nvidia.com/nsight-systems</u>
- User Guide: https://docs.nvidia.com/nsight-systems/UserGuide
- Sample Program (Older Profiler): https://developer.nvidia.com/blog/even-easier-introduction-cuda/