## Profiling GPU codes with Nsight

S H A R C N E T<sup>M</sup>

Sergey Mashchenko syam@sharcnet.ca SHARCNET / Compute Ontario / Alliance

May 18, 2022

## Overview

- Why?
- Where?
- How?
- Demo



# Why to profile GPU codes?

- GPUs are significantly more expensive, and less available, than CPUs
- Not all research codes / algorithms are well suited for GPU acceleration.
- As a consequence, profiling is a critical step in a GPU code development, and should start from the very first kernel you write.

## Tools

- NVIDIA is has produced an retired quite a few GPU profilers.
- The following profilers are no longer maintained (though still available, work up to V100):
  - nvprof: command-line function-level profiler, for both GPU and CPU parts of the code
  - nvvp: graphical (GUI) profiler

## Tools: Nsight

- NVIDIA also maintained for years their Nsight suite of products (IDE / debugger / profiler).
  - Nsight Eclipse edition (Linux, Mac)
  - Nsight Visual Studio Edition (Windows)
- Since 2018, Nsight profilers became also available as three stand alone packages: Nsight Compute, Nsight Systems, and Nsight Graphics.

## Nsight packages



## Where to run Nsight

- If you have a fairly capable recent GPU inside your laptop / PC, you can install and use Nsight suite (Eclipse for Linux/Mac, Visual Studio for Windows) on your own computer.
- But if the goal is to optimize your code for Alliance national systems, you definitely want to run Nsight remotely on our clusters.
  - Command line (CLI) tools can be submitted as jobs
  - Interactive GUI tools can be used on compute nodes allocated with salloc, with either X11 or VNC connections (more about that later).
- If your internet is too slow, you can run CLI Nsight tools on a cluster, then analyze the results on your computer using GUI Nsight.

## How to use GUI tools on clusters

- X11 forwarding (MobaXterm for Windows, Xquartz for Mac): \$ ssh -Y graham.computecanada.ca
   \$ salloc --x11 ...
- VNC on a compute node (requires two terminal windows) [login\_node]\$ salloc ... [gra123]\$ export XDG\_RUNTIME\_DIR=\${SLURM\_TMPDIR} [gra123]\$ vncserver [your\_PC]\$ ssh graham.computecanada.ca -NL 5902:gra123:5901
  - On your PC, launch TigerVNC viewer
  - Enter the destination: localhost:5902

## VNC helper script

#!/bin/bash

```
# Required for vncserver:
export XDG_RUNTIME_DIR=${SLURM_TMPDIR}
```

```
# Starting VNC server, recording the channel:
N=$(vncserver 2>&1 |grep "^New" |cut -d: -f3)
```

```
# Computing the remote port:
Rport=$((5900 + $N))
```

# Printing the command for the local computer: echo "ssh \$USER@\$SLURM\_CLUSTER\_NAME.computecanada.ca -NL 5902:\$SLURMD\_NODENAME:\$Rport"

## NVIDIA profilers on our clusters

- Loaded when a cuda module is loaded, e.g.
   \$ module load cuda/11.4
- Old tools (up to Volta: P100, V100):
  - nvprof (CLI)
  - nvvp (GUI)
- Nsight tools (work on P100\*, V100, T4, A100):
  - ncu, ncu-ui: Nsight Compute (CLI / GUI; V100 and up)
  - nsys, nsys-ui: Nsight Systems (CLI / GUI; P100 and up)

# Compiling code

- Compile the code as usual (after loading the cuda module)
  - The only extra compiler switch required is -lineinfo (do not use -G that one is for debugging)

\$ module load cuda
\$ nvcc -02 -arch=sm\_70 -lineinfo code.cu

## Nsight Compute

• Command: ncu, ncu-ui

\$ ncu -o output\_file code

- Typically the first step in profiling a GPU code
- Allows one to maximize the performance of individual kernels
- Full documentation: https://docs.nvidia.com/nsight-compute/NsightCompute

### Kernel metrics

NVIDIA Nsight Compute			- 🗆 ×
Eile Connection Debug Profile Iools Window Help 🕸 Connect & Disconnect × Terminate 🛛 🖏 Profile Kernel 🖉 🛇 💁 🕸	ੴ →ŧ →• →[ →] ╢ ╠ ᠺ。 ᢗ。 ᢗ。 ║ा		
D Untitled 1 * ×			
Page: Details 🔹 Launch: 2 - 20 - VecAdd_kernel 🔹 🟹 🔹	Add Baseline 👻 Apply <u>R</u> ules		Copy as Image 👻
Current 20 - VecAdd_kernel (196, 1, 1)x Time: 5.02 usecond Cycles: 5,	51 Regs: 16 GPU: NVIDIA GeForce RTX 2080 Ti SM Freq	uency: 999.83 cycle/usecond CC: 7.5 Pro	ocess: [10968] CuVectorAddDrv.exe 🛞 🕞 0
▼ GPU Speed Of Light Throughput			GPU Throughput Chart 🔹 🔎
High-level overview of the throughput for compute and memory resources of the Gi throughput for each individual sub-metric of Compute and Memory to clearly identi	U. For each unit, the throughput reports the achieved perce y the highest contributor. High-level overview of the utilizat	ntage of utilization with respect to the theor tion for compute and memory resources of	retical maximum. Breakdowns show the the GPU presented as a roofline chart.
Compute (SM) Throughput [%]	3.56 Duration [usecond]		5.02
Memory Throughput [%]	16.71 Elapsed Cycles [cycle	e]	5,051
L1/TEX Cache Throughput [%]	8.37 SM Active Cycles [cyc	tle]	2,135.37
L2 Cache Throughput [%]	9.52 SM Frequency [cycle/u	usecond]	999.83
DRAM Throughput [%]	16.71 DRAM Frequency [cycle	e/nsecond]	5.01
	GPU Throughput		
Compute (SM) [%]			
0.0 10.0 20.0 3	.0 40.0 50.0 Speed Of Light (SOL) [%]	60.0 70.0 8	 0.0 90.0 100.0
Compute Workload Analysis			Q
Detailed analysis of the compute resources of the streaming multiprocessors (SM), in overall performance.	cluding the achieved instructions per clock (IPC) and the uti	lization of each available pipeline. Pipelines	with very high utilization might limit the
Executed Ipc Elapsed [inst/cycle]	0.08 SM Busy [%]		6.07
Executed Ipc Active [inst/cycle]	0.20 Issue Slots Busy [%]		6.07
High Pipe Utilization     [Usarning] All pipelines are under-utilized. Either this     details.	۰، ۲۹   kernel is very small or it doesn't issue enough warps per sch	eduler. Check the <u>Launch Statistics</u> and <u>Sch</u>	eduler Statistics sections for further
Memory Workload Analysis			All 👻 🗘
Detailed analysis of the memory resources of the GPU. Memory can become a limitir bandwidth between those units (Max Bandwidth), or by reaching the maximum thro	g factor for the overall kernel performance when fully utilizi ughput of issuing memory instructions (Mem Pipes Busy). D	ng the involved hardware units (Mem Busy) Jetailed chart of the memory units. Detailed	), exhausting the available communication I tables with data for each memory unit.
Memory Throughput [Gbyte/second]	80.38 Mem Busy [%]		9.52
L1/TEX Hit Rate [%]	0 Max Bandwidth [%]		16.71
L2 Hit Rate [%]	40.33 Mem Pipes Busy [%]		3.56

#### Nsight, by Sergey Mashchenko

## Line-by-line info

#### View: Source and SASS 🔻

Neutrol										
Navigation:					Navigation: Sampling Data (All)					
# Course		Sampling	Instructions dic	ated-On Thread	# Address Se		Sampling	Instructions		
* Source	$Ac[\pm v][\pm v] = A[a \pm wA * \pm v \pm \pm v]$	1 220	3 712 000	118 78/ 000	209 0000000 00b9be00		1/	64 000		
	Pe[+v][+v] = P[h + vP + +v + +v],	1,250	2 712 000	119 794 000	210 0000000 00000000	MOV 022 020	10	64 000		
	bs[cy][cx] = b[b + wb + cy + cx],	0	3,712,000	110,704,000	210 00000000 000555210	MOV R25, R20	21	64,000		
90	// Synchronize to make sure the matrice	0			212 00000000 00050220	MOV 021, 022	14	64 000		
	synchronize to make sure the matrice	41	129 000	4 096 000	213 0000000 00050050	MOV 821, 821	250	2 112 000		
	sync tin eaus();	41	120,000	4,030,000	214 00000000 00050000	MOV 022, 022	500	2,112,000		
	// Wultiplu the two estainer together.	0			214 0000000 00050E50	NOV 620, 623	512	2,112,000		
	// multiply the two matrices together;	0			- 215 0000000 00050E00	MOV R22, R22	501	2,112,000		
05	// each thread computes one element	0			217 0000000 00050E70	TEETD LT AND DO DT DOO DT	405	2,112,000		
96 #2222	// OF THE DIOCK SUD-INALITX				217 0000000 00050E80	DLODZ LUT DO DT DO DT DT OVO	495	2,112,000		
07 #prag	gma unroll	0			210 0000000 00050E50	PLOPS.LUT P0, PT, P0, PT, PT, 0x0, 0	495	2,112,000		
		7 542	21 616000	046 176 000	219 0000000 00090Ea0 @	P0 DRA 0x00009C430		2,112,000		
90	for (int $k = 0$ ; $k < BLOCK_SIZE$ ; ++ $k$ ) {	7,542	31,010000	946,176,000	220 0000000 00090E00	BRA 0X00009Dec0	480	2,048,000		
100	CSUD += AS[TY][K] + BS[K][TX];	39,200	101,792,000	5,177,544,000		MOV R21, 000	503	2,046,000		
100		U				MOV R21, R21	504	2,048,000		
		0				MOV R21, R21	322	2,048,000		
	// Synchronize to make sure that the pr	U			224 00000000 0009DET0	MOV R21, R21	490	2,048,000		
	// computation is done before loading t	U			225 00000000 00090+00	MOV R23, R2		2,048,000		
104	// sub-matrices of A and B in the next	0	120.000	1 005 000	226 0000000 00090+10	MOV R26, c[0x0][0x18]		2,048,000		
105	syncthreads();	23	128,000	4,096,000		MOV R27, c[0x0][0x1c]		2,048,000		
106 }	}	0			228 0000000 00090+30	IADD3 R26, P0, R21, R26, RZ	4/0	2,048,000		
107		0			229 0000000b 00b9b+40	IADD3.X R27, R23, R27, RZ, P0, !PT		2,048,000		
108 /	// Write the block sub-matrix to device mem	0			230 0000000 00090+50	MOV R24, R12		2,048,000		
	// each thread writes one element	0			231 0000000 00595460	SHF.R.S32.HI R25, RZ, Øx1f, R24	469	2,048,000		
110 i	<pre>int c = wB * BLOCK_SIZE * by + BLOCK_SIZE *</pre>	2	32,000	1,024,000	232 000000b 00b9b <del>f</del> 70	MOV R24, R24		2,048,000		
111 (	C[c + wB * ty + tx] = Csub;	31	128,000	4,096,000	233 0000000b 00b9bf80	MOV R25, R25	490	2,048,000		
112 }		9	12,800	409,600	234 0000000b 00b9bf90	MOV R21, R24	457	2,048,000		
113		0			235 0000000b 00b9bfa0	MOV R23, R25	467	2,048,000		
11/ unid	ConstantInit/Flast #data int size flast				236 0000000h 00h0hfh0	MOV 001 001	1001	2 040 0		

#### Nsight, by Sergey Mashchenko

# **Nsight Systems**

• Command: nsys, nsys-ui

\$ nsys profile -o output\_file code

- Typically the last step in profiling a GPU code
- Fine-tuning the interactions between kernels, memcopies, CPU code etc, both synchronous and asynchronous
- Full documentation: https://docs.nvidia.com/nsight-systems/UserGuide

### CUDA trace

Timeline View								
0s	+990ms	+995ms	1s	+5ms	+10	ms	+15ms	+20ms
CPU (6)	1204 px; 1 msec		_					
Threads (3)								
iGPU (NVIDIA Tegra X2)								
CUDA (NVIDIA Tegra X2, 0000)	1204 px; 0 msec							
<ul> <li>Default stream (7)</li> </ul>	1204 px; 1 msec							
<ul> <li>Memory</li> </ul>	1204 px; 0 msec							
DtoA memcpy	1204 px; 0 msec	M						
▼ Kernels	1204 px; 2 msec		advectVelocity_k	r	vecto)	)	gul (vector_fft) (u	a
regular_fft	1204 px; 0 msec			r		r	jula)	
vector_fft	1204 px; 1 msec				vecto		vector_fft	
advectVelocity_k	1204 px; 1 msec		advectVelocity_k					
_nv_static_4532_spRea	1204 px; 0 msec			0	0		0	
_nv_static_4532_spRea	1204 px; 1 msec				(			
diffuseProject_k	1204 px; 0 msec				(d)			
_nv_static_4532_spRea	1204 px; 1 msec							
advectParticles_k	1204 px; 0 msec							a
1 kernel group(s) hidden.								

### Live demo

## Reduction code

- The code adds up elements of a long vector on GPU.
- The first version uses a very slow method: serialized summation using atomicAdd() function.
- The second version uses the proper way: parallel summation using binary reduction.

## **Binary reduction**



## Staged copy/compute code

- The first version of the code first copies a long vector to GPU, then carries out independent per-element computations.
- The second version hides some of the costs of copying data to GPU by running copying and computing in parallel, using two GPU streams.

## Staged copy/compute algorithm



You need two streams for this:



### Thank you!