Code profiling on Graham

Sergey Mashchenko (SHARCNET / Compute Ontario / Compute Canada)

Outline

- Introduction
- Simple profiling
- Profiling serial, MPI and OpenMP codes with MAP
- CUDA profiling
- Questions?

Introduction

What is profiling?

- Profiling is the task of timing a code.
- It used used primarily as a part of the iterative process of improving the efficiency (reducing the wallclock runtime) of the code.
- It is often done using simple means (like inserting time measurement lines in your code), but for serious profiling work one has to use dedicated profiling tools.

Simple profiling

Timing the whole code

- On SHARCNET clusters, one can use the Linux shell command "time" to time the whole code.
 - \$ time ./your_code , or
 - \$ time mpirun -np 32 ./your_mpi_code
- This has to be done on an empty node, to improve the accuracy of timing.
- A node can be reserved with salloc command (gives interactive access to compute resources for up to 3 hours), e.g.

\$ salloc --time=0-03:00 -c 32 -A def-**account** --mem=120000M (for serial and multi-threaded codes), and

\$ salloc --time=0-03:00 -n 32 -A def-**account** --mem-per-cpu=4000M (for MPI codes).

Timing the whole code (cont.)

- On Graham, cpu cores take non-negligible time to spin up from the idle state (1200 MHz) to the maximum speed under full load (2600 MHz).
- As a result, one has to time the code multiple times in a loop, choosing the best timing, e.g.:

\$ for ((i=0; i<10; i++)); do { time -p ./code ; } |& grep real ; done |sort -k 2 -gr

- This is obviously not ideal. A much better way is to place timers inside your code, to time specific parts of the code.
- This again should be ideally done in an internal (code) loop, to eliminate the cpu spin-up effect.

Timers inside your code

 gettimeofday: high precision (10µs) cpu-based timer (just google for the function timeval_subtract).

• **OpenMP code:** omp_get_wtime() can be used to time (in seconds) both entire parallel regions, or individual threads inside a parallel region.

• This can also be used to time non-OpenMP codes (as it is more convenient than gettimeofday), just don't forget to add the "#include <omp.h>" line, and compile the code with "-qopenmp" (icc) or "-fopenmp" (gcc) switches.

• MPI code: MPI_Wtime() can be used the same way as omp_get_wtime() on the previous slide:

• **CUDA code:** to time a specific CUDA kernel, the best approach is to use CUDA events:

```
cudaEvent_t start, stop;
float time;
cudaEventCreate (&start);
cudaEventCreate (&stop);
cudaEventRecord (start, 0);
kernel_to_time <<<grid, threads>>> ();
cudaEventRecord (stop, 0);
cudaEventSynchronize (stop);
cudaEventElapsedTime (&time, start, stop);
cudaEventDestroy (start);
cudaEventDestroy (stop);
```

• For timing CUDA code consisting of multiple kernels, and/or concurrent GPU and CPU computations, and/or concurrent GPU operations (using streams), one has to use cpu-based timers (like gettimeofday or omp_get_wtime).

Profiling serial, MPI and OpenMP codes with MAP

Overview

- Parallel profiler MAP (along with the parallel debugger DDT) are now a part of the software package Forge.
- The original company behind MAP was Allinea. In 2016 it was acquired by the CPU maker ARM.
- SHARCNET has been using (and paying for) Allinea/ARM products since 2006.
- Though the debugger DDT was a success from the beginning, the Allinea's first attempt at parallel profiling (OPT) was a failure.
- The replacement MAP (came about in 2013; originally only for serial/MPI codes) used a much better approach to parallel profiling, and is now widely used in HPC community.

Overview (cont.)

- On Graham, profiler MAP is provided via module "ddt-cpu" (or aliases "allinea-cpu" and "arm-forge-cpu").
- The Graham's license is for up to 512 concurrent cpu cores across all users (for both MAP and DDT).
- Niagara cluster (operated by SciNet) has a smaller license (up to 128 cpu cores). Neither MAP nor DDT are available on Cedar.

How to use MAP

- MAP (and DDT) are GUI applications, so one has to enable X11 forwarding in the SSH connection to be able to use them.
 - One has to add "-Y" switch to the usual ssh command:

\$ ssh -Y user@graham.computecanada.ca

- Windows users: use free software MobaXterm, which comes with both SSH client and X window server (required for X11 forwarding).
- Mac users: install free app XQuartz (has X windows server).
- Linux users: everything you need is already installed on your box.
- As an alternative, one could use VNC connection to graham (google for "VNC compute canada" for details).
 - It has a better performance, but takes longer to set up.

How to use MAP (cont.)

• When using the X11 forwarding method, you need to add "--x11" switch to your salloc command, e.g.

\$ salloc --x11 --time=0-03:00 -c 32 -A def-account --mem=120000M

• After allocating the node(s) with salloc, load the module:

\$ module load ddt-cpu/7.1

- The code to be profiled has to be compiled with "-g" switch which adds symbolic information to the binary. You should use all your usual optimization flags (e.g. -O2).
- Run the code under MAP like this:

\$ map ./your_code <optional code arguments>

Some details

- Use the 7.1 version for now, as the newer one has some issues.
- Request one more cpu core than your code needs with salloc, as MAP uses one cpu core at 100% inside salloc session (it is likely a bug; we'll try to fix it).
- If you need more resources than available with salloc (>3h runtime, or hundreds of cpu cores), submit the MAP session as a job, e.g.
 module load ddt-cpu/7.1
 map --profile -n 16 ./code
- The profiling results (*.map files) can be analyzed offline with MAP: \$ map results.map

Application: /project/6000	0898/Profiling/a.out	Details
Application: /project/6000	898/Profiling/a.out	- 🗧
Arguments:		•
□ std <u>i</u> n file:		-
Working Directory:		• 🗧
Duration: Sampling entire	e program	Details
☐ CUDA Kernel analysis		Details
더 MPI: 15 processes, Oper	n MPI	Details
Number of Processes: 15	* *	
□ Processes per Node 1	×	
Implementation: Open MPI	C <u>h</u> ange	
mpirun argumentsovers	subscribe	•
□ Profile selected ranks:	0-14 100%	Select All
□ OpenMP		Details
□ Submit to Queue	Configure Pa	rameters
Environment Variables:	none	Details
<u>H</u> elp O <u>p</u> tions	R	un Cancel

File Edit View Metrics Window Help

Profiled: a.out on 15 processes, 1 node, 30 cores (2 per process) Sampled from: Wed Dec 12 2018 11:15:11 (UTC-05) for 52.8s

Main thread activity	
Memory usage 95.7 89.5 MB	Default
11:15:11-11:16:04 (52.839s): Main thread compute 99.0 %, MPI 1.0 %	Zoom 🕫 🍬 💿 interface
🕫 tsp_mpi2.c 🛙	Time spent on line 225 & ×
<pre>// We always start from the same (0-th) city: perm[0] = 0; // Cycle for Monte Carlo steps: for (k=0; k<n_mc_local; k++)<br="">{ // Generating a random permutation: // Initially we have an ordered list: for (l=1; l<n_cities; l++)<br="">perm[1] = 1; // Then we reshuffle it randomly, starting with l=1: d = 0.0; for (l=1; l<n_cities-1; l++)<br="">{ // This generates a random integer in the range [1 N_CITIES-1]: // This generates a random integer in the range [1 N_CITIES-1]: // This generates a random integer in the range [1 N_CITIES-1]: // This generates a random integer in the range [1 N_CITIES-1]: // This generates a random integer in the range [1 N_CITIES-1]: // State for the state state</n_cities-1;></n_cities;></n_mc_local;></pre>	Breakdown of the 5.4% spent on this line: Executing instructions Calling other functions Time in instructions exo Scalar floating-point Vector floating point Scalar integer Wentor integer Memory access 10 Branch Other instructions
Input/Output Project Files Main Thread Stacks Functions	,

Main Thread Stacks

Total core time $\overline{\nabla}$ MPI	Function(s) on line	Source	Position ^
	🖻 💈 a.out [program]		
	🖻 🖌 main	{	tsp_mpi2.c:146
49.2% australia in a shall all and a shall all	rand_r		tsp_mpi2.c:222
28.8% <u>- the state of the sector of the sect</u>		<pre>d = d + dist[perm[1-1]][perm[1]];</pre>	tsp_mpi2.c:231
5.4%		<pre>ltemp = perm[1];</pre>	tsp_mpi2.c:225
4.7%		<pre>perm[11] = ltemp;</pre>	tsp_mpi2.c:227
3.7%		for (l=1; l <n_cities-1; l++)<="" td=""><td>tsp_mpi2.c:219</td></n_cities-1;>	tsp_mpi2.c:219
1.6%		if (d < d_min_loc)	tsp_mpi2.c:240
Showing data from 15,000 sample	es taken over 15 processes (1000 per process)	Ā	llinea Forge 7.1 💈 Main Thread View

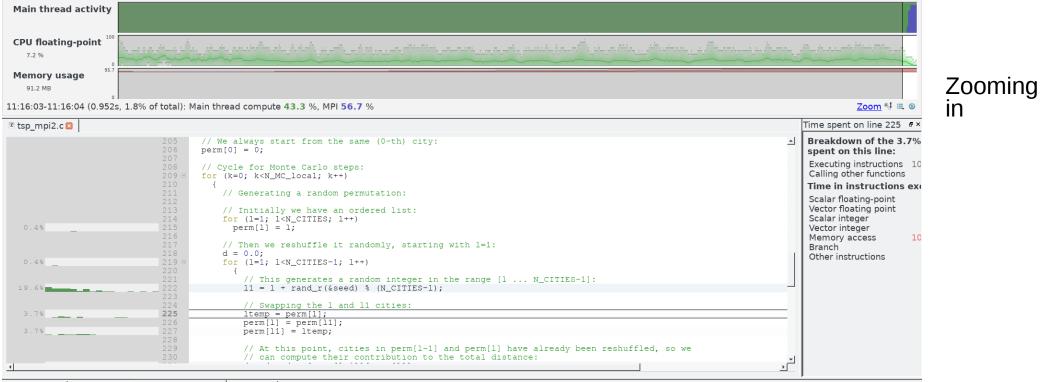
Showing data from 15,000 samples taken over 15 processes (1000 per process)

December 19, 2018

đΧ

File Edit View Metrics Window Help

Profiled: a.out on 15 processes, 1 node, 30 cores (2 per process) Sampled from: Wed Dec 12 2018 11:15:11 (UTC-05) for 52.8s



Input/Output | Project Files | Main Thread Stacks | Functions |

Main Thread Stacks					ēΧ
Total core time ∇ N	MPI	Function(s) on line	Source	Position	^
		🖻 🛸 a.out [program]		-	-
		🖻 🖊 main	{	tsp_mpi2.c:146	
56.7%5	6.7%	MPI_Allreduce	MPI_Allreduce (&result_loc, &result_glob, 1, MPI_FLOAT_INT, MPI_MINLOC, MPI_C.	tsp_mpi2.c:267	
19.6%		≖rand_r		tsp_mpi2.c:222	
11.1%			<pre>d = d + dist[perm[1-1]][perm[1]];</pre>	tsp_mpi2.c:231	
3.7%			<pre>perm[11] = ltemp;</pre>	tsp_mpi2.c:227	
3.7%			<pre>ltemp = perm[1];</pre>	tsp_mpi2.c:225	
2.2%			if (d < d_min_loc)	tsp_mpi2.c:240	-
Showing data from 270 samp	oles take	en over 15 processes (18 per process)	Alli	nea Forge 7.1 💈 Main Thread Vi	ew

December 19, 2018

File	Edit	View	Metrics	Window	Help

Profiled: a.out on 15 processes, 1 node, 30 cores (2 per process) Sampled from: Wed Dec 12 2018 11:15:11 (UTC-05) for 52.8s

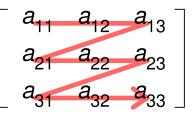
Hide	Metrics.

Main thread activity					
· · · · · · · · · · · · · · · · · · ·					
MPI call duration					
MPI sent 5.11 B/s 0					MPI
MPI received 19.6 5.11 B/s					preset
MPI calls 2.45 0.64 calls/s					
MPI point-to-point					
MPI collectives					
MPI point-to-point bytes					
MPI collectives bytes			_		
11:16:02-11:16:04 (1.480s, 2.8% o	of total): Main thread compute 63.6 %, MPI 36.4	%		Zoom 🍕 🇮 💿	
🕫 tsp_mpi2.c 🛛				Time spent on line 225 🛛 ×	
	205 // We always start from the s 206 perm[0] = 0; 207	ame (0-th) city:	1	Breakdown of the 4.0 - spent on this line:	
0.2%	208 // Cycle for Monte Carlo step 209 B for (k=0; k <n_mc_local; k++)<="" td=""><td>s:</td><td> ▼</td><td>Executing instructions Calling other functions Time in instructions e</td><td></td></n_mc_local;>	s:	 ▼	Executing instructions Calling other functions Time in instructions e	
Input/Output Project Files Mai 1ain Thread Stacks	in Thread Stacks Functions			ت ۲۵ ×	
Fotal core time 🛛 🗸 MPI		Source	Posit	ion 🔶	
36.4% 36.4% 36.4%	■rand_r	{ MPI_Allreduce (&result_loc, &result_glob, 1, MPI_FLOAT_INT, MPI_MINLOC 11 = 1 + rand_r(&seed) % (N_CITIES-1);	C, MPI_C tsp_r tsp_r	mpi2.c:222	
4.0%		<pre>d = d + dist[perm[l-1]][perm[l]]; ltemp = perm[l];</pre>		npi2.c:231 npi2.c:225	
Showing data from 420 samples ta	aken over 15 processes (28 per process)		. • =	rge 7.1 🏼 Main Thread View	

Serial code profiling

- Example: a typical efficiency issue is when a loop in the code reads elements of the vector/array not in the order the data is stored in the memory. This makes CPU-memory caching inefficient.
- For C/C++ codes, data is stored in a row-major order, so it is the last index in multidimensional arrays which should correspond to the inner-most loop:

for (int i=0; i<N; i++) for (int j=0; j<N; j++) A[i][j] = 0.0; Row-major order





Profiled: a.out on 1 process, 1 node Sampled from: Wed Dec 12 2018 11:58:47 (UTC-05) for 32.2s

Hide Metrics...

Main thread activity					
CPU floating-point					
CPU integer 35.0 %					Serial code
CPU memory access				A shake a san that ye	profiling (CPU
CPU fp vector 0 %					instructions preset).
CPU integer vector					 Ded memory
CPU branch 10.5 %		<u>~~_^_~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~</u>	ALA.		Bad memory access case.
11:58:47-11:59:20 (32.221s): Main thread compute 100.0 %			<u>Zoom</u> �I ≡ ⊚	
🕫 bad_cache_use.c 🛛 📔				Time spent on line 21 #×	
32.6%	<pre>13 14 15 16 17 = for (int i=0; i<n; 18="" 19="" 21="" 22="" 23="" 24="" <="" i++)="" pre=""></n;></pre>	The problem	۵ 	Breakdown of the 67. spent on this line: Executing instructions ! Calling other functions 4 Time in instructions e Scalar floating-point Vector floating point Scalar integer Vector integer Memory access	
Input/Output Project Files 1ain Thread Stacks	Main Thread Stacks Functions			ē ×	
	Function(s) on line	Source		ition cache use.c:12	
67.3% 32.6% 0.1%	e rand_r • •1 other	<pre>A[j][1] = (double)rand_r(&seed); for (int j=0; j<n; j++)<="" pre=""></n;></pre>	bac	cache_use.c:21 	
Showing data from 1,000 sa	mples taken over 1 process (1000 per process)	Allinea Forge	e 7.1 🎽 Main Thread View 🏾	
Decembe	r 19, 2018	"Code profiling on Graham" Sergey Mashchenko, SHARCNET			24/39

File	Edit	View	Metrics	Window	Help

Profiled: a.out on 1 process, 1 node Sampled from: Wed Dec 12 2018 12:02:45 (UTC-05) for 7.9s

Hide Metrics...

Main thread activity		
CPU floating-point		
CPU integer 33.0 %		Serial code profiling
CPU memory access ¹⁰⁰		(CPU
CPU fp vector		instructions preset).
CPU integer vector		
CPU branch		Good memory access case.
12:02:45-12:02:53 (7.933s): Main thread compute 100.0 %	Zoom 🍕 🏨 🛛	400000 0400.
🖻 good_cache_use.c 😰	Time spent on line 21 🔮 ×	
<pre>100.0% 10 .</pre>	Breakdown of the 100 [↑] time spent on this line Executing instructions 4 Calling other functions 5 Time in instructions e Scalar floating-point Vector floating point Scalar integer Vector integer Memory access	
Input/Output Project Files Main Thread Stacks Functions Main Thread Stacks	5 ×	
Total core time ∇ Function(s) on line Source	Position	
b main { b frand_r A[1][j] = (double)rand_r(&seed); 53.4% rand_r (no debug info) 4.0% (no debug info) (no debug info)	good_cache_use.c:12 good_cache_use.c:21	
Showing data from 399 samples taken over 1 process (399 per process)	Allinea Forge 7.1 💈 Main Thread View 🏾	
"Code profiling on Cychem"		

December 19, 2018

Profiling MPI codes

- Dynamic workload balancing (DWB) is frequently used by MPI programs.
- We use it when the length of time spent on computing different parts of a large workload by different MPI ranks is hard or impossible to predict ahead of time.
- Well written DWB code should have a way to adjust the size of the workload quantum. (In other words number of chunks.)

Dynamic workload balancing example

- Example code:
 - dynamic_workload_balancing.c: using "nanosleep" function to emulate different processing time for different elements of a large input array
 - On 15 graham cpu cores, I got the following wall clock times:



<u>File Edit View Metrics Window Help</u>

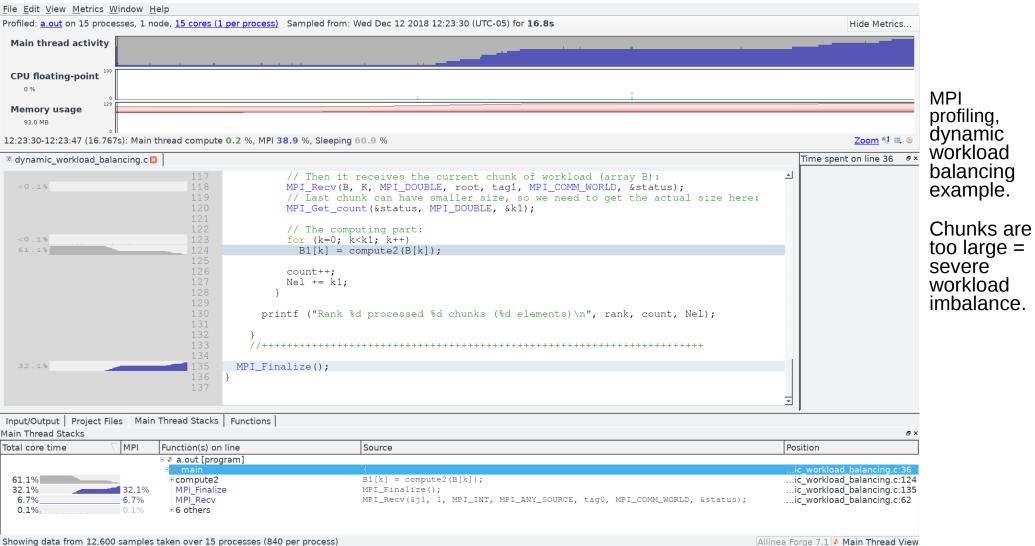
Profiled: a.out on 15 processes, 1 node, 15 cores (1 per process) Sampled from: Wed Dec 12 2018 12:30:22 (UTC-05) for 19.0s

Hide Metrics...

Main thread activity		Menthelist, defidences and an one allowed	
CPU floating-point 100			MPI
Memory usage 90.7 MB 12:30:22-12:30:41 (19.026s): Main thread compute	0 1 % MDI 44 4 % Slooping 55 5 %		profiling, dynamic
	5 0.1 %, Mri 44.4 %, Sleeping 55.5 %	Time spent on line 36 #×	workload
Ø dynamic_workload_balancing.c ☑ 1.5% 117 1.5% 118 119 120 <0.1% 120 55.6% 124 125 125 126 124 127 128 129 130 131 132	<pre>// Then it receives the current chunk of workload (array B): MPI_Recv(B, K, MPI_DOUBLE, root, tag1, MPI_COMM_WORLD, &status); // Last chunk can have smaller size, so we need to get the actual size here: MPI_Get_count(&status, MPI_DOUBLE, &k1); // The computing part: for (k=0; k<k1; k++)<br="">B1[k] = compute2(B[k]); count++; Nel += k1; } printf ("Rank %d processed %d chunks (%d elements)\n", rank, count, Nel); } </k1;></pre>		balancing example. Chunks are too small = latency dominated.
133 134 135 136 137 ▲ Input/Output Project Files Main Thread Stacks	<pre>//+++++++++++++++++++++++++++++++++++</pre>		

Main Thread Stacks đΧ Total core time MPI Function(s) on line Source Position 🗄 🖇 a.out [program] .ic workload balancing.c:36 55.6% B1[k] = compute2(B[k]);...ic_workload_balancing.c:124 compute2 33.0% MPI_Recv(&j, 1, MPI_INT, root, tag3, MPI_COMM_WORLD, &status); ...ic_workload_balancing.c:113_ MPI_Recv 1.8% MPI_Send(&j, 1, MPI_INT, root, tag0, MPI_COMM_WORLD); ...ic_workload_balancing.c:108 1.8% MPI_Send 1.8% 1.8% MPI Recv MPI_Recv(&j1, 1, MPI_INT, MPI_ANY_SOURCE, tag0, MPI_COMM_WORLD, &status); ...ic_workload_balancing.c:62 1.7% MPI Send 1.7% MPI_Send(&A[j], k1, MPI_DOUBLE, source, tag1, MPI_COMM_WORLD); ...ic workload balancing.c:84 1.6% 1.6% MPI Send MPI_Send(&j, 1, MPI_INT, source, tag3, MPI_COMM_WORLD); ...ic workload balancing.c:74 -Showing data from 15,000 samples taken over 15 processes (1000 per process) Allinea Forge 7.1 💈 Main Thread View

December 19, 2018



"Code profiling on Graham" Sergey Mashchenko, SHARCNET

December 19, 2018

<u>File Edit View Metrics Window Help</u>

Profiled: a.out on 15 processes, 1 node, 15 cores (1 per process) Sampled from: Wed Dec 12 2018 12:27:11 (UTC-05) for 11.1s

Hide Metrics.

	10000, <u>10 00100 (1 por p</u>				
Main thread activity					
CPU floating-point		· · · · · · · · · · · · · · · · · · ·			MDI
Memory usage					MPI
91.1 MB					profiling,
12:27:11-12:27:23 (11.127s): Mai	n thread compute 0.2 9	%, MPI 7.4 %, Sleeping 92.4 %		<u>Zoom</u> ♥ ≒ ⊚	dynamic
dynamic_workload_balancing.c				Time spent on line 124 🛛 ×	workload
<0.1%	117 118 119 120 121 122	<pre>// Then it receives the current chunk of workload (array B): MPI_Recv(B, K, MPI_DOUBLE, root, tag1, MPI_COMM_WORLD, &status); // Last chunk can have smaller size, so we need to get the actual size here: MPI_Get_count(&status, MPI_DOUBLE, &k1); // The computing part:</pre>	<u>_</u>	Breakdown of the 0.2% spent on this line: Executing instructions 1 Calling other functions 8 Time in instructions ext Scalar floating-point	balancing example. Chunks
92.6%	123 124	<pre>for (k=0; k<k1; b1[k]="compute2(B[k]);</pre" k++)=""></k1;></pre>		Vector floating point Scalar integer	have just
0.3%	125 126 127 128 129 130 131 132 133 134	<pre>count++; Nel += k1; } printf ("Rank %d processed %d chunks (%d elements)\n", rank, count, Nel); } //++++++++++++++++++++++++++++++++++</pre>		Vector integer Memory access* 10 Branch 6 Other instructions * 40.0% memory access instructions, 60.0% implici memory accesses in other instructions, also counted categories	the right size.
	137				
4			<u>*</u>		
Input/Output Project Files Ma Main Thread Stacks	in Thread Stacks Fund	ctions		r B ×	
Total core time 💎 MPI	Function(s) on line	Source	Pos	sition	
92.6% 6.6% 6.6% 0.8% 0.8%	 a.out [program] main compute2 MPI_Recv 6 others 	{ B1[k] = compute2(B[k]); MPI_Recv(&j1, 1, MPI_INT, MPI_ANY_SOURCE, tag0, MPI_COMM_WORLD, &status);	i	c_workload_balancing.c:36 workload_balancing.c:124 c_workload_balancing.c:62	
Showing data from 14,235 sample	es taken over 15 proces	ses (949 per process)	Allinea Fo	rge 7.1 🍫 Main Thread View	

OpenMP profiling

- In OpenMP, critical regions have a significant overhead and should be used sparingly.
- In particular, if used inside a loop for reduction, there should be a pre-selection statement, outside of the critical region:

if (x > x_max) // Pre-selects only plausible candidates
 #pragma omp critical
 if (x > x_max) // Very infrequently threads would enter the critical
 x_max = x; // region, for the definitive "if" clause application

<u>File Edit View Metrics Window Help</u>

Profiled: a.out on 1 process, 1 node, 15 cores (15 per process) Sampled from: Wed Dec 12 2018 14:34:18 (UTC-05) for 37.7s



<pre>impact.</pre> impact. imp	Application activity CPU floating-point 0% Memory usage 30.4 MB 14:34:18:14:34:55 (37.795): Main thread compute 0.0 % OpenMB 0.8 % Open		Zoom &Lie &	OpenMP profiling, dynamic critical region's						
Input/Output Project Files OpenMP Stacks OpenMP Regions Functions OpenMP Stacks Total core time Overhead Function(s) on line Source Total core time Overhead Function(s) on line Source 90.1% 0.8% 0.8% 0.8% 0.8% 0.8% 0.8% 0.8% 0.8	<pre>13 = int main (int argc, char **argv) 14 14 15 16 17 17 17 17 17 17 17 17 17 17 17 17 17</pre>									
90.1% 90.1% 90.1% 0.8% 0.8% 0.8% openmp_critical.c:28 90.1% 0.8% 0.8% 0.8% 0.8% 0.8% 90.1% 0.8% 0.8% 0.8% 0.8% 0.8% 90.1% 0.8% 0.8% 0.8% 0.8% 0.8% 90.1% 0.8% 0.8% 0.8% 0.8% 0.8% 90.1% 0.8% 0.8% 0.8% 0.8% 0.8% 90.1% 0.8% 0.8% 0.8% 0.8% 0.8% 90.1% 0.8% 0.8% 0.8% 0.8% 0.8% 90.1% 0.8% 0.8% 0.8% 0.8% 0.8% 90.1% 0.8% 0.8% 0.8% 0.8% <th colspan="10">Input/Output Project Files OpenMP Stacks OpenMP Regions Functions</th>	Input/Output Project Files OpenMP Stacks OpenMP Regions Functions									
90.1% 90.1% 90.1% if (r > r_max) 90.1% 90.1% if (r > r_max) 90.1% 0.8% 0.8% 90.1% 0.8%	Total core time $\overline{\nabla}$ Overhead Function(s) on line	Source	Position							
	90.1% 90.1% 0.8% 90.1% 90.	<unknown> [multiple frames] (no debug info) .2.3_2kmpc_loc_pack.43 (no debug info)</unknown>	openmp_critical.c:28							
	showing data nom 1,000 samples taken over 1 process (1000 per process)									

Profiled: a.out on 1 process, 1 node, 15 cores (15 per process) Sampled from: Wed Dec 12 2018 14:37:43 (UTC-05) for 4.6s

Hide Metrics...

openmp_critical.c:28

Allinea Forge 7.1 So OpenMP View

Promed: a.out on 1 process, 1 hode, 15 cores (15 per proce	sampled from: wed Dec 12 2010 14:37:45 (01C-05) for 4.05	Hide Metrics
Application activity		
CPU floating-point		
0%		OpenMP
		profiling,
Memory usage		dynamic
21.9 MB		critical
14:37:43-14:37:47 (4.585s): Main thread compute 0.1 %,	OpenMP 99.3 %, Uncategorized 0.6 %	
🖲 openmp_critical.c 🗵		Time spent on line 24 #× impact.
20 21 22 87.9% 5.8% 87.9% 23 24 24 24 24 25 27 27 28 29 30 31 32 35 35		Breakdown of the 87.9 spent on this line: Executing instructions Calling other functions 10 Pre- selection = good performance.
Input/Output Project Files OpenMP Stacks OpenMP F OpenMP Stacks	Regions Functions	₫×
Total core time V Overhead Function(s) on line	Source	Position
🗄 🕸 a.out [program]		
ie ∕ main ie ≫ main [OpenM	{ [P region 0]	openmp_critical.c:14 openmp_critical.c:17
rand_r		openmp_critical.c:24
81.9%	rand_r (no debug info)	
6.0% 	<unknown> (no debug info) {</unknown>	openmp_critical.c:23
	if(r > r max)	

 if (r > r_max)

-

CUDA profiling

- Recently MAP became capable of CUDA (GPGPU) code profiling.
- Unfortunately, SHARCNET's license doesn't cover this feature.
- But we do have Nvidia provided visual profilers for CUDA programs nvvp and nsight.
- Unfortunately, they don't provide line-by-line kernel analysis (the way MAP does). But they provide plenty of detailed info on kernel performance.
- nvvp and nsight are bundled with cuda modules.

CUDA profiling (cont.)

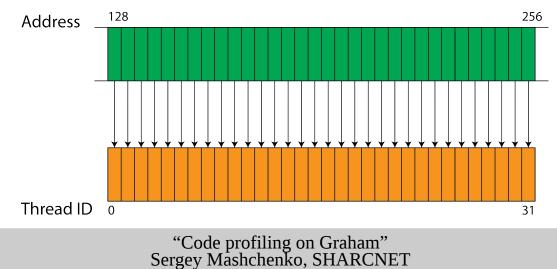
- No code re-compilation is needed for nvvp profiling.
- Using nvvp interactively (on graham and cedar):

\$ salloc --x11 --time=0-03:00 --ntasks=1 --gres=gpu:1
-A def-account --mem-per-cpu=4G
\$ module load cuda/10
\$ nvcc -O2 your_code.c -o your_code
\$ nvvp ./your_code

- The app will provide a step-by-step profiling setup. You choose which kernels to profile, and what specific details you need.
- The app will often provide useful descriptive suggestions regarding which parts of your code have efficiency issues.

CUDA profiling (cont.)

 Non-coalesced access of the device memory is a significant efficiency issue in GPU programming (similar to the row-major memory access requirement on CPUs).



Good (coalesced) access pattern:

December 19, 2018

<u>F</u> ile <u>∨</u> iew <u>W</u> indow <u>R</u> un <u>⊢</u>	lelp														
📑 🕞 🖳 🖳 🖏 🗣 📢	€	🔨 📉 🖪 🔚 🛛 👗	5 -												
💺 *NewSession1 🕱		,													- 0
	0 s 0.05	s 0.1 s	0.15 s	0.2 s	0.25 s	0.3 s	0.35 s	0.4 s	0.45 s	0.5 s	0.55 s	0.6 s	0.65 s	0.7 s	0.75
Process "bad" (20994)	_ <u>`</u> `														
Thread 399721344															
└ Runtime API							cudaLaun	chKernel							
L Driver API															
Profiling Overhead															
[0] Tesla P100-PCIE-12GB															
Context 1 (CUDA)															
Compute														coalesce(void)	
└ 🍸 100.0% coalesc.														coalesce(void)	
Streams															
∟ Default														coalesce(void)	
										_					
										C	Sign of r	non-co	nalo	hang	
										<u> </u>	ngn or i		Juc	SUCU	
										n	nomorv	2000	cc		
											nemory	alle	22		
											-	× 1			
🗔 Analysis 🕱 🛅 GPU Deta	ile (Currennend)	U Deteile 🚍 OpenACC 🛛	Deteile 🗖 Concele	Cottingo					N 6	🗆 🗖 Prope	rtian M				- 0
			Details 🖵 Console	La Settings					N L						
🗉 🗄 🔶 📖 Expo	ort PDF Report	Results								coalesc	e(void)				
1. CUDA Application Ana	lveie	💧 Low Global Men	•		-					Block	Size			[256,1,1]	-
· · ·	-	Global store efficiency of bytes stored divide									ters/Thread			9	
2. Performance-Critical k	Kernels	memory transfers by									ed Memory/Block		· \	08	
The results on the right show		and thus determines	the efficiency of that	at store. Low effic	ciency indicates t	ĥat one or more gl			cess		-				
kernels ordered by potential improvement. Starting with th		pattern or alignment.	. Select this result to	o highlight kerne	is with low global	store efficiency.			More		ch Type			Normal	
highest ranking, you should s	select an entry from	i Kernel Optimiza	ation Priorities							▼ Efficie					
the table and then perform k discover additional optimizati		The following kernels							higher ranked		bal Load Efficiency			Ma	
		kernels (those that a	ppear first in the list) is more likely t	o improve perfori	mance compared t	o lower ranked ke	rnels.		Glo	bal Store Efficiency			(12.5%	
🐴 Perform Kerne	el Analysis	Rank Description								Sh	ared Efficiency			n/a	
Select a kernel from the table at right	t or from the timeline to	100 [1 kernel inst	ances] coalesce(vo	oid)						Wa	rp Execution Efficiency			100%	
enable kernel analysis. This analysis re data, so your application will be run or	nce to collect that data for									No	t-Predicated-Off Warp B	Execution Efficienc	cy .	97.4%	
the kernel if it is not already available.										▼ Occu					
🛺 Perform Additio	nal Analysis										nieved			91.2%	
You can collect additional information with potential performance problems.	to help identify kernels										eoretical			100%	
select any of the new results at right	to highlight the individual													100%	
kernels for which the analysis applies.											d Memory Configuratio				
											ared Memory Executed	3		0 B	
										•					•

<u>File View Window R</u> un	<u>H</u> elp													
📫 📓 🖳 📑 🗣 🗣 🗉	€ € € [8	3 F 🥆 📕	🚊 🚊 l 🙏 🔹											
د ۲. ∗NewSession1 №														- 0
	0 s	0.05 s	0.1 s	0.15 s	0.2 s	0.25 s	0.3 s	0.35 s	0.4 s	0.45 s	0.5 s	0.55 s	0.6 s	0.65 s
Process "good" (24144)														
Thread 1340064640														
Runtime API								cudaLaur	nchKernel					
L Driver API														
Profiling Overhead														
📃 [0] Tesla P100-PCIE-12GB														
Context 1 (CUDA)														
😑 Compute														
∟ 🝸 100.0% coales	c													
Streams														
L Default														
										Goo	nd mer	nory a	22911	
												nory a	50000	
										patt	ern			
											· · · · ·			
												\mathbf{N}		
🗔 Analysis 🛿 🔤 GPU Det	ails (Summary)	🔜 CPU Details 👖	🗑 OpenACC Details	🗐 Console ा Set	tings				N D	🔲 Properties 🛿				- 6
E E 🔶	oort PDF Report	Results								coalesce(void)				
1. CUDA Application An	ahusia		nel Optimization F							Duration				ms (5,101,7
	-	The follo	wing kernels are ord	lered by optimizatio	n importance based	on execution time	and achieved occupa ared to lower ranked l	ncy. Optimization of	higher ranked	Stream			Default	
2. Performance-Critical				inst in the list/ is filo	re likely to improve	performance compa	area to lower ranked r	terniels.		Grid Size			[156250	00,1,1]
The results on the right sho kernels ordered by potentia	w your applicatio		Description							Block Size			[256,1,1	1]
improvement. Starting with I	the kernels with	the 100 l	1 kernel instances] coalesce(void)						Registers/Thre	ad		10	
highest ranking, you should the table and then perform	select an entry kernel analysis t	o								Shared Mem	ory/Block		0.8	
discover additional optimiza	tion opportúnitie	s.								Launch Type			Normal	
- I Perform Kerr	el Analysis													
Select a kernel from the table at rig	-	ato								Global Load	l Efficiency		n/a	
enable kernel analysis. This analysis data, so your application will be run	requires detailed profi	ling								Global Stor	e Efficiency		100%	
the kernel if it is not already availabl	e.									Shared Effi			n/a	
🕕 Perform Additi	onal Analysis										tion Efficiency		100%	
You can collect additional informatio	n to help identify kern	els								Not-Predica	ated-Off Warp Execu	ution Efficiency	97.4%	
with potential performance problem select any of the new results at righ	s. After running this a t to highlight the indiv	nalysis,												
kernels for which the analysis applies	s									Achieved			â 25.3%	
													1000/	·

4

►

Questions?

 You can always contact me directly (syam@sharcnet.ca) or send an email to help@sharcnet.ca.