A GPU Performance Analysis Library providing arbitrary granularity in time and thread count

September 26, 2019

Jim Rosinski
UCAR/CPAESS

Outline

- Summary of GPTL CPU usage/output
- Motivation for GPU extension
- Design Overview
- GPTL mods since 2017
- System software requirements
- User interface/output
- Status/where next

Current CPU functionality

```
  ret = gptlstart ('c_sw_outside')
!$OMP PARALLEL DO PRIVATE (ret)
  do k=1,npz
    ret = gptlstart_ ('c_sw')
    call c_sw (. . .)
    ret = gptlstop ('c_sw')
  end do
  ret = gptlstop ('c_sw_outside')
  . . .
  ret = gptlpr_file ("timing.0")         ! Print summary stats
  ret = gptlpr_summary (MPI_COMM_WORLD) ! Summarize across tasks
```

- Library is thread-safe => OK to call inside threaded regions
- Single character string to start/stop pairs
- Output routines summarize performance information across threads and/or MPI tasks

CPU results display

```
Stats for thread 0:     Called   Wallclock max       min
  TOTAL                      1   168.263   168.263   168.263
    fv_dynamics             96   104.178     1.204     1.064
      FV_DYN_LOOP          100   107.332     1.193     1.049
        DYN_CORE           200    93.638     0.594     0.457
          c_sw_outside    1200     8.184     0.023  6.24e-03
            c_sw         12492     7.844     0.013  4.40e-04

Same stats sorted by timer for threaded regions:
Thd      Called  Recurse Wallclock max       min
000 c_sw    12492    -       7.844     0.013  4.40e-04
001 c_sw    12498    -       7.844     0.013  4.25e-04
002 c_sw    12395    -       7.798     0.013  4.43e-04
003 c_sw    12603    -       7.881     0.022  4.21e-04
004 c_sw    12764    -       7.939     0.013  4.24e-04
005 c_sw    12848    -       7.981     0.013  4.29e-04
SUM c_sw    75600    -      47.287     0.022  4.21e-04
```

- Indentation shows nested regions
- Also per-thread timings for multi-threaded regions

Motivation/Requirements for GPU timing library

- Need to gather performance info at finer granularity than individual kernels
- Want load balance info across warps for each timed region
- GPU code is in addition to CPU => can have both in a single execution
  - Easy to assess kernel launch overhead
- Minimize timer overhead
- Retain simple API requiring only user addition of start/stop calls
- Must be callable from OpenACC
  - Fortran module (“use gptl_acc”) and C/C++ headers (“#include <gptl_cuda.h>. Both are very simple small files

Requirements for GPU port of GPTL

- Underlying timing routine:
  - nvcc provides clock64()
- Ability to mix CUDA , OpenACC, and C/C++/Fortran
  - GPTL-GPU guts are CUDA, CPU portion is C
  - Fortran wrappers for start/stop timers and output
- Ability to keep separate timers for separate threads
  - Store timers one per warp
  - Linearize the warp number across threads, blocks, and grids

Design Overview

1. Allocate space for 2-d array (warp x timername) to store timing data. Done once per run, via cudaMalloc() from CPU. Max number of warps and max number of timernames are user specifiable.
2. For each timername, generate an integer “handle” index into 2-d array before any start/stop calls are issued. “handle” index is required by start/stop routines.
3. Start/stop timer calls must generate a “linearized” warp number. 3 thread Idx + 3 block Idx. Only thread 0 of each warp is considered.
4. Given warp and timername indices, start/stop functions accumulate stats similar to CPU code. CUDA cycle counter routine clock64() drives the timing calculations.
5. Timing results passed back to CPU for analysis (e.g. #calls, #warps participating, max/min, warp responsible for max/min), and printing.

GPTL mods since 2017

- ”malloc” no longer called anywhere on GPU
  - Use cudaMalloc from host. Required user setting of number of warps, timers on startup
  - 8 MB malloc limit on device no longer an issue
- No string functions for expensive GPTL functions which run on GPU (e.g. GPTLstart, GPTLstop)
  - str* calls are VERY expensive on GPU
  - User must invoke “init_handle” routine for each timer before use

System Software Requirements

- CUDA rev at least 10.0. Others may be OK.
  - Current work used 10.0 (PC) and 10.1 (HPC system)
- PGI rev. at least 18.3. Others may be OK.
  - Current work used 19.4
- NOTE: PGI compute capability needs to match CUDA compute capability
  - Current work had been done with cc60

Limitations of nvcc

- No string functions (strcmp, strcpy, etc.)
  - Roll your own (ugh)
- No realloc()
- No varargs()
- No sleep(), usleep()
- Very limited printing capability
  - printf() OK
  - No fprintf(), sprintf()
- Not C99 compliant => cannot dimension input arrays using input arguments

Code example mixing timing calls for both CPU and GPU

```
use gptl
use gptl_acc
!$acc routine (doalot_log) seq
 integer :: total_gputime, doalot_log_handle

! Define handles
!$acc parallel private(ret)  copyout (total_gputime, doalot_log_handle)
  ret = gptlinit_handle_gpu ('total_gputime'//char(0), total_gputime)
  ret = gptlinit_handle_gpu ('doalot_log'//char(0),    doalot_log_handle)
!$acc end parallel

  ret = gptlstart ('doalot')
!$acc parallel loop private (niter, ret) &
!$acc&  copyin (n, innerlooplen, total_gputime, doalot_log_handle)
  do n=0,outerlooplen-1
    ret = gptlstart_gpu (total_gputime)
    ret = gptlstart_gpu (doalot_log_handle)
    vals(n) = doalot_log ()
    ret = gptlstop_gpu (doalot_log_handle)
    ret = gptlstop_gpu (total_gputime)
  end do
!$acc end parallel
  ret = gptlstop ('doalot')
```

Printed results from code example

Workload increasing from thread 0 through thread 3583:

CPU Results:

	Called	Wall	max	min
total_kerneltime	3	1.401	1.000	1.72e-04
donothing	1	1.64e-04	1.64e-04	1.64e-04
doalot	1	0.401	0.401	0.401
sleep1ongpu	1	1.000	1.000	1.000

GPU Results:

name	calls	warps	holes	wallmax	(warp)	wallmin	(warp)
total_gputime	336	112	0	1.379	111	1.004	0
donothing	112	112	0	2.44e-06	65	2.21e-06	11
doalot_sqrt	112	112	0	0.058	111	5.30e-04	0
doalot_sqrt_double	112	112	0	0.122	111	1.06e-03	0
doalot_log	112	112	0	0.100	111	8.62e-04	0
doalot_log_inner	11200	112	0	0.100	111	9.47e-04	0
sleep1	112	112	0	1.000	99	1.000	5

Printed results from code example

Workload evenly distributed across 3584 threads:

CPU Results:

	Called	Wall	max	min
total_kerneltime	3	1.405	1.000	1.91e-04
donothing	1	1.81e-04	1.81e-04	1.81e-04
doalot	1	0.405	0.405	0.405
sleep1ongpu	1	1.000	1.000	1.000

GPU Results:

name	calls	warps	holes	wallmax	(warp)	wallmin	(warp)
total_gputime	336	112	0	1.379	42	1.379	55
donothing	112	112	0	2.18e-06	97	1.99e-06	7
doalot_sqrt	112	112	0	0.058	98	0.058	48
doalot_sqrt_double	112	112	0	0.122	46	0.122	68
doalot_log	112	112	0	0.100	8	0.100	57
doalot_log_inner	11200	112	0	0.100	54	0.100	97
sleep1	112	112	0	1.000	60	1.000	34

Example from a "real" OpenACC code: NIM weather forecast model

```
subroutine vdmints3(...)
use gptl
use gptl_acc
integer, save :: vdmints3_handle, ipn_handle, ...
logical, save :: first = .true.

if (first) then
  first = .false.
!$acc parallel private(ret) copyout(vdmints3_handle, ...)
  ret = gptlinit_handle_gpu ('vdmints3', vdmints3_handle)
  ret = gptlinit_handle_gpu ('vdmints3_ipn', ipn_handle)
  ...
!$acc end parallel
end if
!$acc parallel private(ret) copyin(vdmints3_handle)
  ret = gptlstart_gpu (vdmints3_handle)
!$acc end parallel

!$acc parallel private(ret) &
!$acc&  num_workers(PAR_WRK) vector_length(VEC_LEN), &
!$acc&  copyin(ipn_handle, kloop1_handle, ...)

!$acc loop gang worker private(rhs1,rhs2,rhs3,Tgt1,Tgt2,Tgt3)
do ipn=ips,ipe
  ret = gptlstart_gpu (ipn_handle)
  ret = gptlstart_gpu (kloop1_handle)
  do k=1,NZ-1
    <...> ! do a bunch of work for each "k"
  enddo !k-loop
  ret = gptlstop_gpu (kloop1_handle)
  ret = gptlstart_gpu(scalar_handle)
  <...> ! do a bunch of work for k=NZ-1
  ret = gptlstop_gpu (scalar_handle)

  ret = gptlstart_gpu(solvei_handle)
  CALL solveiThLS3(nob,nbf,rhs1,rhs2,rhs3,amtx1(1,1,ipn))
  ret = gptlstop_gpu(solvei_handle)

  ret = gptlstart_gpu(isn1_handle)
  do isn = 1,nprox(ipn)
    do k=1,NZ-1
    <...> ! do a bunch of work for each "k"
    enddo
  end do
  ret = gptlstop_gpu(isn1_handle)

  ret = gptlstart_gpu(isn2_handle)
  do isn = 1,nprox(ipn)
    isp=mod(isn,nprox(ipn))+1
    ret = gptlstart_gpu (scalar_handle)
    <...> ! do a bunch of work for k=1 and k=NZ
    ret = gptlstop_gpu (scalar_handle)
  end do
  ret = gptlstop_gpu(isn2_handle)

  ret = gptlstart_gpu(k4_handle)
  do k=1,NZ-1
    <...> ! do a bunch of work for each "k"
  end do
  ret = gptlstop_gpu(k4_handle)

  ret = gptlstart_gpu(scalar_handle)
  <...> ! do a bunch of work for k=0 and k=NZ
  ret = gptlstop_gpu (scalar_handle)
  ret = gptlstop_gpu (ipn_handle)
enddo
!$acc end parallel
!$acc parallel private(ret)
ret = gptlstop_gpu (vdmints3_handle)
!$acc end parallel
end subroutine vdmints3
```

Timing output from “real” OpenACC code

GPU timings:

name	calls	warps	holes	wallmax	(warp)	wallmin	(warp)
vdmints0	4.10e+07	10242	20482	0.282	6	0.154	30723
vdmints3	4000	1	30723	25.987	0	25.987	0
vdmints3_ipn	4.10e+07	10242	20482	0.827	1008	0.501	30723
vdmints3_kloop1	4.10e+07	10242	20482	0.040	3021	0.023	30717
vdmints3_kloop2	0	0	30724	0.00e+00	0	0.00e+00	0
vd3_k3	0	0	30724	0.00e+00	0	0.00e+00	0
vd3_k4	4.10e+07	10242	20482	0.021	4029	8.60e-03	30717
vdmints3_isn1	4.10e+07	10242	20482	0.099	4035	0.039	30723
vdmints3_isn2	4.10e+07	10242	20482	0.437	7053	0.211	30723
vdmints3_scalar	3.28e+08	10242	20482	0.211	7050	0.114	30723
vdmints3_solvei	4.10e+07	10242	20482	0.229	261	0.067	30723
force	4000	1	30723	0.985	0	0.985	0
force_ipn	4.10e+07	10242	20482	0.057	23208	0.025	2676

Timing output from “real” OpenACC code disabling interior vdmints3 calls

GPU timings:

name	calls	warps	holes	wallmax	(warp)	wallmin	(warp)
vdmints0	4.10e+07	10242	20482	0.283	90	0.156	30723
vdmints3	4000	1	30723	17.348	0	17.348	0
vdmints3_ipn	0	0	30724	0.00e+00	0	0.00e+00	0
vdmints3_kloop1	0	0	30724	0.00e+00	0	0.00e+00	0
vdmints3_kloop2	0	0	30724	0.00e+00	0	0.00e+00	0
vd3_k3	0	0	30724	0.00e+00	0	0.00e+00	0
vd3_k4	0	0	30724	0.00e+00	0	0.00e+00	0
vdmints3_isn1	0	0	30724	0.00e+00	0	0.00e+00	0
vdmints3_isn2	0	0	30724	0.00e+00	0	0.00e+00	0
vdmints3_scalar	0	0	30724	0.00e+00	0	0.00e+00	0
vdmints3_solvei	0	0	30724	0.00e+00	0	0.00e+00	0
force	4000	1	30723	0.986	0	0.986	0
force_ipn	4.10e+07	10242	20482	0.057	23160	0.024	2340

Overhead CPU vs GPU

CPU:

```
Underlying timing routine was gettimeofday.
Total overhead of 1 GPTLstart + GPTLstop call=6.6e-08 seconds
Fortran layer:              0.0e+00 =   0.0% of total
Get thread number:          2.0e-09 =   3.0% of total
Generate hash index:        1.0e-08 =  15.2% of total
Find hashtable entry:       8.0e-09 =  12.1% of total
Underlying timing routine: 4.2e-08 =  63.6% of total
Misc start/stop functions: 4.0e-09 =   6.1% of total
```

GPU:

```
Total overhead of 1 GPTLstart_gpu + GPTLstop_gpu pair call=3.0e-06 seconds
Get warp number:                 6.6e-07 =  27.9% of total
Underlying timing routine+SMID: 7.2e-08 =   3.1% of total
Misc calcs in GPTL_start_gpu:    6.7e-07 =  28.4% of total
Misc calcs in GPTL_stop_gpu:     9.6e-07 =  40.6% of total
```

Status/Where Next

- Merge in autoconf build structure from trunk (hard)
- Add MPI summary info as has been done for CPU (hard)
- Report grid(x,y,z), block(x,y,z) and thread(x,y,z) info rather than linearized warp (easy)
- Performance optimizations?
- GNU compiler support for OpenACC?
- Adding call tree structure (indentation) likely won't happen due to computational cost
- Auto-profiling possible, requires compiler flag to auto-generate profiling entry points (-finstrument-functions on CPU)

Testers/helpers welcome

- Open source GPTL library is available at:
  - https://github.com/jmrosinski/GPTL
- Branch “cuda_acc” enables GPU capability
- Trunk is CPU-only
- What other features are needed?
- Are the existing features useful?