CERE hands-on training

https://benchmark-subsetting.github.io/cere/

2017-03-21

The goal of this training is to learn how to use CERE to extract codelets from a parallel application. Here we will use the LULESH proxy application developed at Lawrence Livermore National Lab. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh.

We will show how extracted codelets can be used for fast OpenMP strong scalability analysis and compiler and parameter tuning.

1 Preliminary setup

1. First, we allocate a dedicated node on the jetson-tx mini-cluster to improve performance measures stability.

```
user@jetson-tx01:~$ salloc --nodes=1 --time=1:00:00 --partition=jetson-tx --cpu-freq=1734000 srun
 ↪  --pty bash
salloc: Granted job allocation 25057
salloc: Waiting for resource configuration
salloc: Nodes jetson-tx02 are ready for job
user@jetson-tx02:~$
```

2. Load CERE with the following commands,

```
$ source /apps/modules/3.2.10/Modules/default/init/bash
$ module load CERE
load binutils/2.26 (PATH, MANPATH, LD_LIBRARY_PATH)
load gcc/5.2.0 (PATH, MANPATH, LD_LIBRARY_PATH)
load libunwind/git (LD_LIBRARY_PATH, LIBUNWIND_INCL, LIBUNWIND_LIBS)
load CERE/0.2.2 (PATH, MANPATH, LIBRARY_PATH, LD_LIBRARY_PATH, LD_RUN_PATH, PYTHONPATH)
```

3. Check that CERE is working,

```
$ cere -h
usage: cere [-h] [--version]
      {configure,profile,capture,replay,check-matching,select-max-cov,select-ilp,
      instrument,trace,check,regions,report,selectinv,flag,hybrid}
      ...
```

4. Get LULESH from `/home/cakel/lulesh2.0.3.tgz`,

```
cp /home/cakel/lulesh2.0.3.tgz ~/.
tar -xvf ~/lulesh2.0.3.tgz
```

5. CERE capture and replay invoke specific LLVM compiler passes. To easily compile an application, CERE includes a compiler wrapper, `cerec`. You can either use `cerec` directly to compile and link a program, or modify the Makefile so it uses `cerec`.

```
- SERCXX = g++ -DUSE_MPI=0
+ SERCXX = cerec -DUSE_OMP=1 -DUSE_MPI=0
- CXX = $(MPICXX)
+ CXX = $(SERCXX)
+ LD = cerec
- LDFLAGS = -g -O3 -fopenmp
+ LDFLAGS = -O3 -fopenmp -lstdc++ -L/apps/gcc/5.2.0/lib64/
```

6. Set environment variables. Select number of threads for capture,

```
$ export OMP_NUM_THREADS=2
```

Select affinity,

```
$ export KMP_AFFINITY="scatter,1"
```

Choose CERE warm-up strategy for replay. CERE provides 3 warm-up options: `COLD`, no warm-up at all; `WORKLOAD`, the whole workload is touched before replaying; `PAGETRACE`, the trace of most recently touched pages is loaded before replay,

```
$ export CERE_WARMUP="WORKLOAD"
```

Select the number of meta-repetitions at replay,

```
$ export CERE_REPLAY_REPETITIONS=1
```

2 Find and capture regions of interest with CERE

1. We need to tell CERE which commands to use for building and running the application. Call `cere configure` with the following arguments,

```
$ cere configure --build-cmd="make -j4" --run-cmd="./lulesh2.0 -s 15" --clean-cmd="make clean && rm
↪  -f *.ll" --omp
```

`cere configure` saves the project configuration in the file `cere.json`. You can manually edit this file if you wish to change the initial values.

2. To find the regions of interest, CERE profiles the application and measures the contribution of each region,

```
$ cere profile --regions
```

`cere profile` determines the percentage of execution time for each extractible region. It also captures the region call graph, `.pdf` and `.dot` versions of the call graph can be found in `.cere/profile/`.
One can list the extractible regions with the following command,

```
$ cere regions
```

which outputs the file `regions.csv` detailing for each region: its name, its location, and its coverage informations.

3. As you can see in `regions.csv`, the main region is `__cere__lulesh__ZL28CalcFBHourglassForceForElemsR6DomainPdS1_S1_S1_S1_S1_S1_dii_810` which covers around 5% of the total application runtime. In the following we will focus on this particular region.
Since the region name is quite long, we can export it in an environment variable,

```
$ export ROI="__cere__lulesh__ZL28CalcFBHourglassForceForElemsR6DomainPdS1_S1_S1_S1_S1_S1_dii_810"
```

Before using a region in a simulation or optimization study, one must check that the replay actually matches the original behavior. Otherwise what you observe during replay may not be what is truly happening in the original execution of the application.

The following command will automatically test this region,

```
$ cere check-matching --region=$ROI
```

Cere check-matching automatically traces the region performance, selects representative invocations, captures and replays them. The final output of the command above should be something like,

```
Results for region:
 ↪  __cere__lulesh__ZL28CalcFBHourglassForceForElemsR6DomainPdS1_S1_S1_S1_S1_S1_dii_810
  MATCHING: In vitro = 12819227.2287 & invivo = 11212240.0 (error = 12.5357574216%, coverage =
   ↪  5.2%)
      Invocation 51: In vitro cycles = 31462.0 & in vivo cycles = 27518.0 (error = 12.5357574217%,
       ↪  part = 407.451122901)
```

The codelet version of this region is faithful to the original one with an error of 12.5%. CERE needs only to replay one invocation to predict the execution time [1].

3 Performance prediction

CERE highlight is that using codelets is usually much faster than running the full application. For this training, we intentionally took a quite small dataset so that the capture and the profile are quick. Replaying a single region in this case, can be comparable to running the full application. On the other hand with the default input, the application execution time is about 97.52 seconds. The region above covers around 15% of the application and it takes 5.1 seconds to replay it as a codelet. Using the codelet with the standard dataset is therefore 19 times faster.

3.1 Strong scalability prediction

CERE OpenMP codelets enable strong scalability prediction; that is to say knowing how the performance of a parallel region will behave while changing the number of threads with a fixed dataset. This is faster with codelets since we do not have to execute the full application.

1. We can replay the previous region with 4 threads, only 3 invocations are executed to predict the total region runtime.

```
$ export OMP_NUM_THREADS=4
$ cere replay --region=$ROI -f
Predicted cycles for region:
 ↪  __cere__lulesh__ZL28CalcFBHourglassForceForElemsR6DomainPdS1_S1_S1_S1_S1_S1_dii_810
 Invocation 51: In vitro cycles = 16598.0 (part = 407.451122901)
 Overall predicted cycles = 6762873.73791
```

   CERE predicts that this region will run in 6.76×10^6 cycles with 4 threads which is 1.7 times faster than the 2 threaded version.

2. Is this prediction accurate? To compare, we can measure the execution time of the region inside the application with 4 threads,

[1] In Arm64, cycles are measured with the counter-timer virtual count register (CNTVCT_EL0)

```
$ cere instrument --region=$ROI
$ cat __cere__lulesh__ZL28CalcFBHourglassForceForElemsR6DomainPdS1_S1_S1_S1_S1_S1_dii_810.csv
Codelet Name,Call Count,CPU_CLK_UNHALTED_CORE
__cere__lulesh__ZL28CalcFBHourglassForceForElemsR6DomainPdS1_S1_S1_S1_S1_S1_dii_810,400,6362422
```

With 4 threads, the original region is executed in 6.36×10^6 cycles. CERE prediction is accurate with an error of 6.2%

3. What happens if you try to predict the performance for 1 thread? (Repeat steps 1 and 2 with `OMP_NUM_THREADS=1`). For the answer follow the footnote[2].

3.2 Compilation flags performance prediction

Trying multiple compilation flags combinations is tedious since you have to run the full application for each change. As for the scalability prediction, we can use the codelet to quickly evaluate the impact of changing compilation flags.

1. To predict the performance of this region when compiled in `-O0`. Edit the Makefile and change `-O3` by `-O0` and run the replay

```
- CXXFLAGS = -g -O3 -fopenmp -I. -Wall
+ CXXFLAGS = -g -O0 -fopenmp -I. -Wall

- LDFLAGS = -O3 -fopenmp -lstdc++ -L/apps/gcc/5.2.0/lib64/
+ LDFLAGS = -O0 -fopenmp -lstdc++ -L/apps/gcc/5.2.0/lib64/
```

```
$ cere replay --region=$ROI -f
...
Overall predicted cycles = 21270170.9688
```

CERE predicts a performance of 2.13×10^7 cycles, 3.1 times slower than the 4 threads, `-O3` version.

2. We can again compare to the original performance to be sure that CERE prediction is accurate

```
$ cere instrument --region=$ROI
$ cat __cere__lulesh__ZL28CalcFBHourglassForceForElemsR6DomainPdS1_S1_S1_S1_S1_S1_dii_810.csv
Codelet Name,Call Count,CPU_CLK_UNHALTED_CORE
__cere__lulesh__ZL28CalcFBHourglassForceForElemsR6DomainPdS1_S1_S1_S1_S1_S1_dii_810,400,20322555
```

Once again CERE successfully predicts the performance with an error of 4.4%

4 Conclusion

We only used a small part of CERE possibilities. For instance, CERE provides commands to automatically select a set of regions that best represents the application. These codelets can be selected either to maximize the coverage of matching codelets or to minimize replay time. CERE can also generate an `HTML` report that summarize selection results. For more information you can check cere manual pages,

```
$ man cere tutorial
```

The report for lulesh2.0 -s 30 (default dataset size) on Jetson-TX and Intel Haswell is available at `https://benchmark-subsetting.github.io/cere/#Others`.

[2]It should not work. Indeed lulesh uses an optimized specific code path when running with 1 thread. Since we captured the region with 2 threads, CERE is executing a different code path at replay. This is one of the limitations of CERE OpenMP replay; if the code path changes depending on the number of threads, the replay may not be faithful.