Codee Training Series
April 26-27, 2022

NERSC

codee

Shift Left Performance
Automated Code inspection for Performance
Walkthrough Exercise: Calculating $\pi$ on the GPU with OpenMP/OpenACC

Goals
- Produce OpenACC version for GPU
- Produce OpenMP version for GPU
- Build & run an OpenMP code on the GPU (for problem size $N=900000000$)
- Build & run an OpenACC code on the GPU (for problem size $N=900000000$)
### The GPU programming challenges: Example code PI

<table>
<thead>
<tr>
<th>Example codes used in this introductory course</th>
<th>Challenges of GPU acceleration addressed in introductory course</th>
<th>Other GPU programming challenges to be addressed in next advanced course</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Find opportunities for offloading</td>
<td>Optimize memory layout for data transfers</td>
</tr>
<tr>
<td>PI</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>MATMUL</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LULESHmk</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>HEAT</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Your code!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <steps>\n", argv[0]);
        printf("  <steps> controls the precision of the approximation.\n");
        return 0;
    }

    // Reads the test parameters from the command line
    unsigned long N = atol(argv[1]);
    printf("- Input parameters\n");
    printf("steps\t= %lu\n", N);

    printf("- Executing test...\n");
    double time_start = getClock();
    // ============================================================

    double out_result;
    double sum = 0.0;
    for (int i = 0; i < N; i++) {
        double x = (i + 0.5) / N;
        sum += sqrt(1 - x * x);
    }
    out_result = 4.0 / N * sum;
    // ============================================================

    double time_finish = getClock();
    // Prints an execution report
    printf("time (s)= %.6f\n", time_finish - time_start);
    printf("result\t= %.8f\n", out_result);
    const double realPiValue = 3.141592653589793238;
    printf("error\t= %.1e\n", fabs(out_result - realPiValue));

    return 0;
}
Inspecting the code and optimizing its performance with Codee

1. Get the performance optimization report for the whole code base
2. Create performance-optimized code for the hotspot automatically

- src
  - Profiling tool (e.g. GNU gprof)
  - hotspots
  - preport
  - performance report
  - preport
  - Directives code (OpenMP, OpenACC, ICC, Clang)

Repeat until the target performance is achieved (% runtime reduction, speedup)
# Produce the entry-level report for default #actions

(pwreport --evaluation)

```bash
$ pwreport --evaluation pi.c
Target Lines of code Analyzed lines Analysis time # actions Effort Cost Profiling
----------------- -------------- ------------- -------------- --------- ------ ------ ---------
pi.c            43            4              33 ms         2         16 h   523€   n/a

ACTIONS PER OPTIMIZATION TYPE
Target Serial scalar Serial control Serial memory Vectorization Multithreading Offloading
----------------- -------------- ------------- ------------- ------------- -------------- ----------
pi.c            0             0              0             2             n/a            n/a

Target : analyzed directory or source code file
Lines of code : total lines of code found in the target (computed the same way as the sloccount tool)
Analyzed lines : relevant lines of code successfully analyzed
Analysis time : time required to analyze the target
# actions : total actionable items (opportunities, recommendations, defects and remarks) detected
Effort : estimated number of hours it would take to carry out all actions (serial scalar, serial control, serial memory, vectorization, multithreading and offloading with 1, 2, 4, 8, 12 and 16 hours respectively)
Cost : estimated cost in euros to carry out all the actions, paying the average salary of 56,286€/year for a professional C/C++ developer working 1720 hours per year
Profiling : estimation of overall execution time required by this target

SUGGESTIONS
You can specify multiple inputs which will be displayed as multiple rows (ie. targets) in the table, eg:
pwreport --evaluation some/other/dir pi.c

Use --actions to find out details about the detected actions:
pwreport --actions pi.c

Multithreading and offloading actions are filtered by default. Use --include-tags to enable them:
pwreport --include-tags all pi.c

You can focus on a specific optimization type by filtering by its tag (serial-scalar, serial-control, serial-memory, vectorization, multithreading, offloading), eg:
pwreport --actions --include-tags serial-scalar pi.c

1 file successfully analyzed and 0 failures in 33 ms
```

By default multithreading and offloading are disabled in Codee.

Rationale: Codee forces the user to explicitly enable multithreading and offloading capabilities to avoid common errors resulting from a misconfigured software environment (e.g., lack of an OpenMP compiler with offload).
2: Produce the entry-level report for ALL #actions

(pwreport --evaluation --include-tags all)

$ pwreport --evaluation --include-tags all pi.c

Target Lines of code Analyzed lines Analysis time # actions Effort Cost    Profiling
--------- ----------- -------------- ------------- --------- ------ ------- ---------
pi.c 43            4              34 ms         4         44 h   1439€   n/a

ACTIONS PER OPTIMIZATION TYPE
Target Serial scalar Serial control Serial memory Vectorization Multithreading Offloading
--------- -------------- -------------- -------------- -------------- -------------- ----------
pi.c 0             0              0             2             1              1

Target : analyzed directory or source code file
Lines of code : total lines of code found in the target (computed the same way as the sloccount tool)
Analyzed lines : relevant lines of code successfully analyzed
Analysis time : time required to analyze the target
# actions : total actionable items (opportunities, recommendations, defects and remarks) detected
Effort : estimated number of hours it would take to carry out all actions (serial scalar, serial control, serial memory, vectorization, multithreading and offloading with 1, 2, 4, 8, 12 and 16 hours respectively)
Cost : estimated cost in euros to carry out all the actions, paying the average salary of 56,286€/year for a professional C/C++ developer working 1720 hours per year
Profiling : estimation of overall execution time required by this target

SUGGESTIONS
You can specify multiple inputs which will be displayed as multiple rows (ie. targets) in the table, eg:
pwreport --evaluation some/other/dir --include-tags all pi.c

Use --actions to find out details about the detected actions:
pwreport --actions --include-tags all pi.c

You can focus on a specific optimization type by filtering by its tag (serial-scalar, serial-control, serial-memory, vectorization, multithreading, offloading), eg:
pwreport --actions --include-tags serial-scalar pi.c

1 file successfully analyzed and 0 failures in 34 ms

By enabling ALL actions in the report now identifies 1 offload opportunity
3: Produce the report of ALL #actions per type of loops
(pwreport --evaluation --include-tags all --level 2)

```
$ pwreport --evaluation --level 2 --include-tags all pi.c
Target Lines of code Analyzed lines Analysis time # actions Effort Cost Profiling
pi.c 43 4 33 ms 4 44 h 1439€ n/a

ACTIONS PER OPTIMIZATION TYPE
Target Serial scalar Serial control Serial memory Vectorization Multithreading Offloading
pi.c 0 0 0 2 1 1

ACTIONS PER LOOP TYPE PER OPTIMIZATION TYPE
Loop Type No. Loops Serial scalar Serial control Serial memory Vectorization Multithreading Offloading
Low 1 0 0 0 2 1 1
Medium 0 0 0 0 0 0 0
High 0 0 0 0 0 0 0

Target : analyzed directory or source code file
Lines of code : total lines of code found in the target (computed the same way as the sloccount tool)
Analyzed lines : relevant lines of code successfully analyzed
Analysis time : time required to analyze the target
# actions : total actionable items (opportunities, recommendations, defects and remarks) detected
Effort : estimated number of hours it would take to carry out all actions (serial scalar, serial control, serial memory, vectorization, multithreading and offloading with 1, 2, 4, 8, 12 and 16 hours respectively)
Cost : estimated cost in euros to carry out all the actions, paying the average salary of 56,286€/year for a professional C/C++ developer working 1720 hours per year
Profiling : estimation of overall execution time required by this target

SUGGESTIONS
You can specify multiple inputs which will be displayed as multiple rows (ie. targets) in the table, eg:
  pwreport --evaluation some/other/dir --include-tags all pi.c
Use --actions to find out details about the detected actions:
  pwreport --actions --include-tags all pi.c
You can focus on a specific optimization type by filtering by its tag (serial-scalar, serial-control, serial-memory, vectorization, multithreading, offloading), eg:
  pwreport --actions --include-tags serial-scalar pi.c

1 file successfully analyzed and 0 failures in 33 ms
```

By increasing the details of the report, the tool reports that ALL the actions are identified in the scope of loops that have LOW difficulty from the performance optimization viewpoint.
4: Produce the Codee Actions Report for the target function

```
pwreport --actions
```

```
FUNCTION BEGIN at pi.c:main:12:1
  12: int main(int argc, char *argv[]) {
    LOOP BEGIN at pi.c:main:31:5
      31:    for (int i = 0; i < N; i++) {
    [RMK011] pi.c:31:5 the vectorization cost model states the loop might benefit from explicit vectorization
    [OPP001] pi.c:31:5 is a multi-threading opportunity
    [OPP002] pi.c:31:5 is a SIMD opportunity
    [OPP003] pi.c:31:5 is an offload opportunity
      LOOP END
    FUNCTION END
  }
CODE COVERAGE
Analyzable files:            1 / 1     (100.00 %)
Analyzable functions:        1 / 1     (100.00 %)
Analyzable loops:            1 / 1     (100.00 %)
Parallelized SLOCs:          0 / 25    (  0.00 %)
METRICS SUMMARY
Total recommendations:         0
Total opportunities:           3
Total defects:                 0
Total remarks:                 1
SUGGESTIONS
Use --level 0|1|2 to get more details, e.g:
pwreport --level 2 --actions
3 opportunities for parallelization were found in your code, get more information with pwloops:
pwloops pi.c:main
More details on the defects, recommendations and more in the Knowledge Base:
https://www.appentra.com/knowledge/
1 file successfully analyzed and 0 failures in 34 ms
```

Each action is reported in the scope of the corresponding loop:
- vectorization (loop:31 OPP002 related to RMK011)
- multithreading (loop:31 OPP001)
- offloading (loop:31 OPP003)
5: Produce the detailed actions for the target function
(pwreport --actions --level 2)

$ pwreport --actions --level 2 --include-tags all pi.c:main

FUNCTION BEGIN at pi.c:main:12:1
8: int main(int argc, char *argv[]) {

LOOP BEGIN at pi.c:main:31:5
31: for (int i = 0; i < N; i++) {
32: double x = (i + 0.5) / N;
33: sum += sqrt(1 - x * x);
34: }

[OPP003] pi.c:31:5 is an offload opportunity
Compute patterns:
- 'scalar' over the variable 'sum'

SUGGESTION: use pwloops to get more details or pwdirectives to generate directives:
pwloops pi.c:main:31:5
pwdirectives --omp offload pi.c:main:31:5 --in-place
pwdirectives --acc pi.c:main:31:5 --in-place

More information on: https://www.appentra.com/knowledge/opportunities

LOOP END
FUNCTION END

By enabling the detailed report for OPP003 (offload opportunity) you obtain suggestions to invoke pwdirectives for automatic annotation of the source code with OpenMP and OpenACC offload directives (note: source code edited "in-place" by default)
6a: Annotate the code for GPU + OpenMP

(pwdirectives --omp offload)

Just copy & paste the suggested invocation of pwdirectives, which will rewrite the code for you adding OpenMP directives

(note: source code edited "in-place" by default and in this example we are using "-o" to write a separate source code file)

By default the OpenMP generated code:
- offloads the computation with “target teams”
- manages data transfers with “map”
- splits workload with “schedule(static)”
6b: Annotate the code for GPU + OpenACC

(*pwdirectives --acc*)

```bash
$ pwdirectives --acc pi.c:main:31:5 -o pi_acc.c
```

Results for file 'pi.c':

Successfully parallelized loop at 'pi.c:main:31:5' [using offloading without teams]:

- **INFO** pi.c:31:5 Parallel scalar reduction pattern identified for variable 'sum' with associative, commutative operator '+'.
- **INFO** pi.c:31:5 Available parallelization strategies for variable 'sum':
  - #1 OpenACC scalar reduction (* implemented)
  - #2 OpenACC atomic access
- **INFO** pi.c:31:5 Parallel region defined by OpenACC directive 'parallel'
- **INFO** pi.c:31:5 Loop parallelized with OpenACC directive 'loop'
- **INFO** pi.c:31:5 Data region for host-device data transfers defined by OpenACC directive 'data'

Successfully created pi_acc.c

Minimum software stack requirements: OpenACC version 2.0 with offloading capabilities

```c
// ================================================
double out_result;
double sum = 0.0;
#pragma acc data copyin(N) copy(sum)
{
#pragma acc parallel
{
#pragma acc loop reduction(+: sum)
for (int i = 0; i < N; i++) {
    double x = (i + 0.5) / N;
    sum += sqrt(1 - x * x);
}
} // end parallel
} // end data
out_result = 4.0 / N * sum;
// ================================================
```

In a similar manner, for OpenACC just copy & paste the suggested invocation of `pwdirectives`, which will rewrite the code for you adding OpenACC directives (note: source code edited "in-place" by default and in this example we are using "-o" to write a separate source code file)

By default the OpenACC generated code:
- offloads the computation with "parallel"
- manages data transfers with "data copy"

(note: OpenACC provides a more elegant solution to manage data transfers for double** data types)
7: Benchmarking on Perlmutter @NERSC (using Nvidia toolchain)

By default, the recommendation for Perlmutter @NERSC is to use the Nvidia Programming Environment.

PI code runs correctly on the GPU @perlmutter and 5.1x faster using OpenMP offload.

PI code runs correctly on the GPU @perlmutter and 7.3x faster using OpenACC offload.

Launch script “launch.sh”

#!/bin/bash
#SBATCH -A ntrain2_g
#SBATCH --reservation=codee_day1
#SBATCH -c gpu
#SBATCH -q regular
#SBATCH -t 0:10:00
#SBATCH -N 1
#SBATCH --ntasks-per-node=1
#SBATCH --gpus-per-task=1
export SLURM_CPU_BIND="cores"
srun PI.sh

PI execution script “PI.sh”

#!/bin/bash
rm pi pi_ompOff pi_acc
nvc pi.c -lm -fast -o pi
./pi 900000000
nvc -mp=gpu -fast -gpu=cc80 -lm pi_ompOff.c -o pi_ompOff
./pi_ompOff 900000000
nvc -acc -fast -gpu=cc80 -lm pi_acc.c -o pi_acc
./pi_acc 900000000

PO: Benchmarking on Perlmutter @NERSC (using Nvidia toolchain)

$ nvc pi.c -lm -fast -o pi
$ ./pi 900000000
  - Input parameters
  steps = 900000000
  Executing test.
  time (s)= 0.873033
  result = 3.14159265
  error = 8.0e-15

$ nvc -mp=gpu -fast -gpu=cc80 -lm pi_ompOff.c -o pi_ompOff
$ ./pi_ompOff 900000000
  - Input parameters
  steps = 900000000
  Executing test...
  time (s)= 0.172202
  result = 3.14159265
  error = 8.9e-14

$ nvc -acc -fast -gpu=cc80 -lm pi_acc.c -o pi_acc
$ ./pi_acc 900000000
  - Input parameters
  steps = 900000000
  Executing test...
  time (s)= 0.119455
  result = 3.14159265
  error = 1.3e-14
Final remarks about using Codee at NERSC

- First, remember to load the Codee module
  ```bash
  $ module load codee
  ```
- The flag `--help` lists all the options available in the Codee command-line tools
  ```bash
  $ pwreport --help
  $ pwloops --help
  $ pwdirectives --help
  ```
- You can run Codee command-line tools on the login nodes (no need to run them on the compute nodes)
- Build and run the example codes on the compute nodes using the batch scripts
  - Scripts tuned to use the appropriate reservations: `codee_day1`, `codee_day2`
- Remember to check the open catalog of rules for performance optimization:
  ```
  https://www.codee.com/knowledge/
  ```