Parallelware Tool Workshop

Learning parallelization of real applications from the ground-up

Manuel Arenaz | October 17, 2019

NERSC

appentra
make code parallel

OpenACC
More Science, Less Programming

OpenMP
Enabling HPC since 1997

Agenda

8:15 - 8:45	*Morning refreshment and coffee*
8:45 - 9:00	*Welcome and introductions*
9:00 - 9:30	Lecture 1: An introduction to OpenMP/OpenACC optimizations for CPUs/GPUs
9:30 - 10:15	Lecture 2: Patterns to minimize data transfers, optimize memory usage and exploit massive parallelism
10:15 - 10:30	Break
10:30 - 11:00	**Lecture 3: Minimizing data transfers**
11:00 - 11:30	Lecture 4: Optimizing memory usage
11:30 - 12:00	Lecture 5: Exploiting massive parallelism
12:00 - 13:00	Working lunch (hands-on activities)
13:00 - 14:00	Practical 5A: Parallelizing the calculation of HEAT
14:00 - 17:00	Hands-on time with your code
17:00 pm	*Close*

Use cases: Performance optimization on CPU/GPU

Use case #1: Minimizing data transfers

- **Code patterns: flow patterns (eg. convergence loop)**
- On GPUs: Transfer data from CPU to GPU and reuse it!
- On CPUs: Create threads and reuse them!

Use case #2: Optimizing memory usage

- **Code patterns: memory patterns (eg. data structure design)**
- On GPUs: Watch your data structure design as it may break your code!
- On CPUs: Hardware keeps memory consistency, so focus mostly on locality!

Use case #3: Exploiting massive parallelism

- **Code patterns: computation patterns (eg. collapsible nested loops)**
- On GPUs: Scale-up to thousands of threads!
- On CPUs: Limited number of threads, so not so important as on GPUs!

Why using flow patterns?

1: Flow patterns provide a deeper understanding of the reuse of data during the execution of the parallel code

- Typically, scientific and engineering codes perform simulations over time where many program inputs are read-only data that does not change at run-time. In GPU programming it is recommended to transfer such data to GPU memory only once at the beginning of the program.

2: Flow patterns enable to the detection of loops that cannot be parallelized

- Time-step loops that dictate the progress in time during the execution of the code cannot be parallelized because, given an initial state of a variable, such variable is updated in each time-step using as inputs the values computed in the previous iteration(s). Thus, all the threads either on the CPU or on the GPU must go through all the time-steps and synch at the beginning/end of each time-step iteration.

Flow Patterns

convergence loop

```
for(iter=0, err = tol; err >= tol && iter < iter_max; iter++){
  ...
}
```

propagation loop

```
for(iter=0; iter < iter_max; iter++){
  ...
}
```

Convergence loop

Understanding the sequential code

- A time-step loop which stops when a fixed maximum number of loop iterations is achieved or when the value of a numerical error metric is less than a fixed threshold.
- At a given step of a convergence loop, the numerical error is computed by combining the solution in the current loop iteration with the solution in previous iterations (typically 1-2 previous iterations).
- As a result, the convergence loop cannot be parallelized because there are dependencies between consecutive loop iterations.

```
C:
int iter = 0; double err;
for(iter=0, err = tol; err >= tol && iter < iter_max; iter++){
  // compute new solution A_iter using as input A_previous_iter
  // compute numerical error between A_iter and A_previous_iter
  // Copy contents of A_iter into A_previous_iter to prepare for next iteration
}

Fortran:
integer iter = 0
real err = 0
do while (iter < iter_max .and. err > tol)
  // compute new solution A_iter using as input A_previous_iter
  // compute numerical error between A_iter and A_previous_iter
  // Copy contents of A_iter into A_previous_iter to prepare for next iteration
end do
```

Propagation loop

Understanding the sequential code

- A simplified version of convergence loops also typically iterates until a maximum number of iterations is achieved.
- Although no numerical error threshold is checked, the solution in the current loop iteration is computed using the solution in previous iterations (typically 1-2 previous iterations).
- As a result, a propagation loop cannot be parallelized because there are dependencies between consecutive loop iterations.

```
C:
int iter = 0;
for(iter=0; iter < iter_max; iter++){
  // compute new solution A_iter using as input A_previous_iter
  // Copy contents of A_iter into A_previous_iter to prepare for next iteration
}


Fortran:
integer iter = 0
do while (iter < iter_max)
  // compute new solution A_iter using as input A_previous_iter
  // Copy contents of A_iter into A_previous_iter to prepare for next iteration
end do
```

Parallelizing the calculation of HEAT on the GPU with OpenMP/OpenACC

Walkthrough:

- Using Parallelware Trainer in function *compute()*:
  - Generate two separate *data* directives for two consecutive loops.
  - Generate one single data directive that covers two consecutive loops.
  - Open the solution with one joined *data* directive with array shapes.
- Using Parallelware trainer in function *cfd_heat_diffusion()*:
  - Open the solution with one single data directive that covers the convergence loop.