Parallelware Tool Workshop

Learning parallelization of real applications from the ground-up

Manuel Arenaz | October 17, 2019

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!

Computation pattern Forall in nested loops

Understanding the sequential code

- A loop that updates the elements of an array.
- Each iteration updates a different element of the array.
- The result of computing this pattern is an array that is the "*output variable*".

parallel forall

```
for (j=0; j<n; j++ ) {
   A[j] = B[j];
}
```

Identifying opportunities for parallelization in different source code variants

```
for (j=0; j<n; j++ ) {
   A[j] = B[j];
}
```

Simple loop

```
for (j=0; j<n; j++ ) {
  for (k=0; k<n; k++ ) {
     A[j][k] = B[j][k];
  }
}
```

Two perfectly nested loops

```
for (j=0; j<n; j++ ) {
  C[j] = 0;
  for (k=0; k<n; k++ ) {
     A[j][k] = B[j][k] + A[j][k-1];
  }
}
```

Not perfectly nested loops: dependencies between iterations

Clause: `collapse`

- Collapses multiple loops into one "larger" loop.
- Use case: when one or more loops are perfectly nested and there are not dependencies that prevent the parallel execution of all of the iterations of all of the loops at the same time.

```
#pragma acc parallel loop collapse(2)
for (j=0; j<n; j++ ) {
  for (k=0; k<n; k++ ) {
     A[j][k] = B[j][k];
  }
}
```

Identifying opportunities for parallelization in different source code variants

```
for (j=0; j<n; j++ ) {
   A[j] = B[j];
}
```

Simple loop

```
for (j=0; j<n; j++ ) {
  for (k=0; k<n; k++ ) {
     A[j][k] = B[j][k];
  }
}
```

OK **collapse(2)**

Two perfectly nested loops

```
for (j=0; j<n; j++ ) {
  C[j] = 0;
  for (k=0; k<n; k++ ) {
     A[j][k] = B[j][k] + A[j][k-1];
  }
}
```

FAIL **collapse(2)**

Not perfectly nested loops: dependencies between iterations

Parallelizing MATrix MULtiplication on the GPU with OpenMP/OpenACC

Walkthrough:

- Using Parallelware Trainer in file version *matmul_v1*, function *matmul()*:
  - Generate one single *data* directive that covers two consecutive loops.
  - In each *loop* directive, add *collapse(2)* clause to exploit parallelism across the two nested loops at the same time.
  - Finally, add an <u>incorrect</u> *collapse(3)* clause for the three nested loops.