Supporting Data Parallelism in Matcloud: Project Report II

Tasks Completed:

Up to now we have finished several prerequisite tasks in our Matlcloud framework, such as designing and implementing the for/parfor node for the abstract syntax tree, and finalized our algorithm to generate code for parfor blocks. We are in the middle of implementing the parfor block code generation. Tasks being finished include:

- Parsing for/parfor blocks and convert them into AST trees. (XW)
- Adding code to launch nvcc and load dynamic libraries at run-time. (YZ)
- Designing code generation algorithms for parfor blocks. (XW & YZ)
- Generating host-side CUDA code. (XW & YZ)

Challenges and Solutions:

The biggest challenges come from analyzing the code inside parfor block and generate CUDA source code for them. Though Matlab data types can only be matrices, their dimensions do not need to be declared before usage. Therefore we need to pre-walk through the parfor block to determine the maximum possible dimension of variables on the destination set and pre-allocate them before launching the CUDA kernel calls. For example, for the parfor code section below, take the first statement as an example, we need to associate the induction variable i of vector a with its maximum value 100 to determine the memory requirement of vector a is 100 float/double numbers. However, for the second statement, the maximum index of vector a is seen when i equals its minimum value 0. Therefore, for each assignment, we need to correlate the induction variable with its minimum and maximum values and execute the AST of the vector index expression to determine the maximum possible memory requirement. Moreover, we further need to walk through every statement in the parfor body and compare their memory requirement, so that we can allocate enough memory for every statement. For the example below, only after the third statement can we determine the maximum memory required is 10,000 when i equals 100.

```
parfor i=0:2:100
  a(i) = i;
```

```
  a(100-i) = i;
  a(i*i) = 0;
endfor
```

Based on above considerations, we finalized our algorithm to generate code for parfor blocks, shown with the pseudo code below. We first find the loop indexes for the parfor loop ("i" in the previous example). We then analyze the memory requirement, as discussed above, and generate code for each statement. In the prototype, for a single parfor loop, we creates a grid of blocks with fixed size 256. Each GPU threads represents one valid value of the induction variable in the loop. To map the thread Id into loop indexes, we generate code in the following template

```
int gid = threadIdx.x + blockIdx.x * blockDim.x;
int i = starting_value + gid * step_size;
```

where starting_value and step_size are 0 and 2 extracted from $i = 0 : 2 : 100$.

As for the host function signature, we need to pass all buffer pointers to the destination and source variables appeared in the block, as well as their dimension sizes. These sizes are used for address calculation for multi-dimensional matrixes.

```
find loop indexes and their ranges
foreach line of code do
    if not an assignment
         continue; // non-assignment commands are ignored
analyse the maximum index;
generate code for the destination sub-tree;
generate code for the source sub-trees;
endfor

generate the host code;
compile as a dynamic library with nvcc
allocate memory for destination matrixes
load dynamic library at run-time
call generated function
```