Notes on DAG software

Mike Giles

December 30, 2013

1 User guide

From the user's point of view, there are just two functions, a host routine `dag_setup` and an inline device function `dag_new_task`.

int* dag_setup(int ntasks, int *input_ptrs, int *input_deps)
This host routine is used to define a DAG by specifying the input dependencies in standard CSR (compressed sparse row) format. It returns a pointer to a GPU device array which the user must pass through to the CUDA kernel.

`ntasks`	number of tasks in the DAG
`input_ptrs`	array of length `ntasks+1`; the first `ntasks` elements give the starting index in `input_deps` for each task, and the final element gives the length of the array `input_deps` (which is also the index of the first element after the end of `input_deps`)
`input_deps`	input dependencies for tasks

int dag_new_task(int* dag, int task_id)
This device function updates the internal DAG info in response to the task which has just been completed, and then determines the next task to be executed. It returns 1 if there is a new task, and 0 if there are none remaining.

`dag`	array pointer generated by dag_setup
`task_id`	on input, the ID of the previous task executed (-1 if there was none); on output, the ID of the next task to be executed. **IMPORTANT**: this **must** be a shared memory variable

Table 1: Input dependencies (tasks which must be executed earlier because they supply the inputs) and output dependencies (tasks which must be executed later because they rely on the outputs) for the test application.

tasks	input dependencies	output dependencies
0		2
1		3, 4
2	0	4, 5
3	1	5
4	1, 2	
5	2, 3	

It is best to look at the test application to fully understand the inputs for `dag_setup`, and the use of the two routines. The input dependencies of its DAG are given in the second column of Table 1. In the code, these input dependencies are represented using the standard CSR format by:

```
int input_ptrs[] = {0, 0, 0, 1, 2, 4, 6};
int input_deps[] = {0, 1, 1, 2, 2, 3};
```

For example, `input_ptrs[4]=2` and `input_ptrs[5]=4` so `input_deps[2]`, `input_deps[3]` give the input dependencies `1, 2` for task `4`.

The host routine `dag_setup` returns the device array pointer `dag` which is then passed into the user's kernel, `my_dag_kernel`. Note that in `my_dag_kernel` the variable `t` which holds the task ID is declared to be a shared memory variable, and it is initialised to -1. The main loop in the kernel is very simple, with the `while` loop continuing for as long as `dag_new_task` returns the value `1` to indicate that there is another task to be executed.

NOTE: I could easily modify `dag_setup` so that it uses the output dependencies rather than the input dependencies, or alternatively accepts a list of dependencies (the "edges" of the graph) each defined by a pair of tasks (the "vertices" of the graph).

2 Implementation

When there are `nt` tasks and `ndeps` dependencies, the `dag` array has the following components:

- `dag[0]` – number of tasks `nt`
- `dag[1]` – atomic counter for tasks launched so far
- `dag[2]` – atomic counter for next empty scheduled task slot
- `dag[3]-dag[nt+2]` – list of scheduled tasks
- `dag[nt+3]-dag[2*nt+2]` – unsatisfied input dependencies
- `dag[2*nt+3]-dag[3*nt+3]` – index for output dependencies
- `dag[3*nt+4]-dag[3*nt+3+ndeps]` – list of output dependencies

The basic idea is that tasks are scheduled for execution once all of their input dependencies have been satisfied. Some tasks have no input dependencies, and so they are immediately put into the scheduled task list by the initialisation in `dag_setup`. For the others, the number of unsatisfied input dependencies is decremented by one each time one of their input tasks is completed, and when it reaches zero the task is scheduled.

The output dependencies are stored in `dag` in the standard CSR format, with an index for each task into the list of output dependencies.

The steps in the host routine `dag_setup` are:

- create a list of dependencies (expressed as pairs of tasks) and sort by input task to obtain an ordered list of output dependencies.
- initialise atomic counters to zero, and set all elements of the scheduled task list to `-1` to indicate there are no scheduled tasks
- for each task, set the number of unsatisfied input dependencies, and if it is zero add the task to the scheduled task list and increment `dag[2]`
- set the output dependency info in `dag`
- transfer the DAG info to the GPU and return the pointer to the GPU device array

The first part of the device function `dag_new_task` is executed only if there has been a previous task executed, in which case it loops over the task's output dependencies and for each one

- decrements the input dependency counter
- if it is now zero, it schedules the task for execution, and increments `dag[2]`

The second part of `dag_new_task` increments the atomic counter `dag[2]` to find the next task for execution in the scheduled task list. If there is a task, it checks whether it has a valid non-negative ID. If it doesn't, then it means that a task has not yet been scheduled in that slot, and so it loops until one is scheduled. It then sets the task ID, and returns a `0` or `1` value depending whether or not there is a task to be executed.