cuRVE: The CUDA Runtime Variable Environment

http://github.com/FLAMEGPU/cuRVE

The University Of Sheffield.

Dr Paul Richmond
p.richmond@sheffield.ac.uk
http://www.paulrichmond.staff.shef.ac.uk

Overview and Simple Example

The CUDA Runtime Variable Environment (curRVE) is a library which provides key-value memory management and access for CUDA global device memory.

GPU device memory can be registered and values set in host code using a constant string expression key via the the cuRVE API. For example the following initialises and sets the value of three cuRVE variables.

```
curveInit(VECTOR_ELEMENTS);
curveRegisterVariable("a");
curveRegisterVariable("b");
curveRegisterVariable("c");

for (int i=0; i<VECTOR_ELEMENTS; i++){
    float a = rand()/(float)RAND_MAX;
    float b = rand()/(float)RAND_MAX;
    curveSetFloat("a", i, a);
    curveSetFloat("b", i, b);
  }
```

A corresponding kernel can be defined using the cuRVE variable access functions as follows;

```
__global__ void vectorAdd()
{
    float a, b, c;
    a = getFloatVariable("a");
    b = getFloatVariable("b");
    c = a + b;
    setFloatVariable("c", c);
}
```

The advantage of cuRVE are;

- A high level method for CUDA Memory management:
  - Data transfer to and from the device is handled transparently.
  - A minimum kernel performance overhead is introduced.
- Simplified Kernel launching:
  - CUDA kernels can be specified and compiled without kernel arguments simplifying kernel launching.
- De-coupling of kernel and host code
  - Using CUDA object linking, kernels and host code can be compiled a linked without any dependencies (any miss matched variables are reported at runtime).
  - Only a single cuRVE header (“curve.h”) is required in kernel or host source files.

Namespaces and Restricting Variable Access

Namespaces allow a cuRVE variable to have a limited scope. Similarly, variable names can be re-used under different namespaces. The cuRVE namespace can be changed using an API call as follows;

```
curveChangeNamespace("vector_addition_example");
```

Namespaces can be used to limit a variable to a particular CUDA kernel (or set of kernels). Further restrictions on variable access may be placed by enabling or disabling variable access as follows;

```
curveDisableVariable("d");
curveEnableVariable("a");
```

Implementation Details

The cuRVE library uses string hashing to map string variable name keys to device memory pointers at runtime. A simple non-cryptographic hash function (FNV 1a) is used to recursively hash string characters through a series of multiplications with primes and bitwise operations. Hash collisions are resolved using linear probing. The hash table is stored locally in host memory and in constant GPU memory ensuring hash lookups are efficient.

Optimisation
The performance of string hashing is greatly improved through template meta programming to recursively expand constant string hashing to a set of static function calls[1]. This enables the compiler to optimise constant string literals (used for variable names and namespaces) to a single 32 bit integer value.

In order to optimise linear probing of hash collisions for kernel string variable lookups, the maximum number of cuRVE variables is always a power of 2 to enable a bitwise modulus implementation. The hash collision loop is implemented using the following for loop allowing loop unrolling. This avoids the slow branching of a naive implementation.

[1] http://www.altdevblogaday.com/2011/10/27/quasi-compile-time-string-hashing/

```
__device__ __inline__ CurveVariable
getVariableLoopUnrolled(const CurveVariableHash variable_hash)
{
   for (unsigned int x=0; x< CURVE_MAX_VARIABLES; x++)
   {
      const CurveVariable i = (variable_hash+x) &
                              (CURVE_MAX_VARIABLES-1);
      const CurveVariableHash h = d_hashes[i];
      if ( h == variable_hash)
         return i;
   }
   return -1;
}
```

Application to FLAME GPU Multi-agent Simulator

The FLAMEGPU multi-agent simulator uses code generation templates to generate complete CUDA host and device code from a simulation model description. This is compiled and linked with user defined agent (kernel) functions to provide a simple mechanism to describe agent behaviour. In its current form FLAMEGPU has the following problems will be solved by the cuRVE library.

Reliance on Kernel Arguments
User defined agent functions require a number of arguments to be passed in a specific order (matching the function call from the code generation process). This is often a cause of errors which are difficult to interpret for non GPU programmers. The cuRVE library can easily be used to retrieve agent and message variables without kernel argument passing.

Unnecessary Data Transfer
FLAMEGPU loads all agent memory data into structures. The use of structures prevents users from accessing data belonging to other agents but requires that all agent memory data is loaded for each agent function. It is often the case that agent functions require only a subset of the agents memory variables. The cuRVE library will ensure that only data required by the agent function will be loaded or stored. Further to this the restriction of variable access will allow agent functions with no dependencies on memory variables to be executed concurrently introducing further parallelism.

Code Repetition
The use of code generated data structures whilst beneficial for understanding code requires that library functions for common agent functionality (e.g. communicating via messages, processing dead agents, etc.) contain large amounts of code repetition. Using cuRVE a single implementation of library functions can be used with variables qualified by the use of namespaces.