Performance Analysis and Improvement of MASS CUDA

Ahmed Musazay
Undergraduate Research

University of Washington, Bothell
Faculty Advisor: Munehiro Fukuda, Ph.D.

What is MASS?

Multi-Agent Spatial Simulation (MASS) is a computing library that allows users to write simulation programs that take advantage of parallelization and distributed systems without knowing the implementation details. As a high level abstraction, it allows users without a computing background to run simulations using multiple computing nodes and multi-threading. There are currently three versions of MASS: C++, Java, and CUDA.

Users define Place and Agent objects. Place objects are simulation entities that contain information and communicate with each other. Agents are subsets of Places, and can communicate, migrate to different Places, and contain information. For example, a program simulating the spread of disease among population can define communities as Places and individuals as Agents.

MASS-CUDA

CUDA, a C/C++ extension by NVidia, is a parallel computing platform and programming interface allowing developers to use the GPU for general purpose processing.

MASS CUDA implements the MASS library functions dealing with intensive computing using the GPU (instantiating objects, calling methods on each object, exchanging data between all objects).

The figure below shows the architecture of MASS CUDA at a high level:

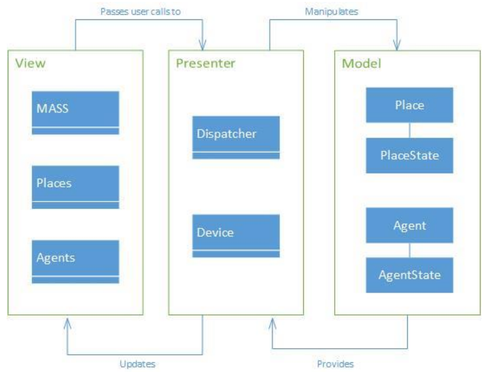

Figure 1: Diagram of high-level architecture of MASS CUDA

The view component provides interaction with the user- instantiating and managing collections of Places/Agents. Calls made by user are then taken to the presenter component, which manages computation logic and handles CUDA function calls. The presenter also manages the Place/State entities, which exist on the GPU device memory and host memory, through the Model layer.

Problem

Although MASS CUDA allows for hundreds of threads executing concurrently, the performance results of the library were quite slow. Figure 2 shows the comparison between a simulation with two different types of programs- one using sequential C++ (no GPU or MASS), and the other using MASS CUDA to run the simulation.

The goals of this project were to analyze in-depth the performance of the library through unit testing of different components (instantiation, callAll function, exchangeAll function), and through overall performance testing using the Heat2D simulation as a benchmark. After a thorough performance analysis, possible solutions were proposed.

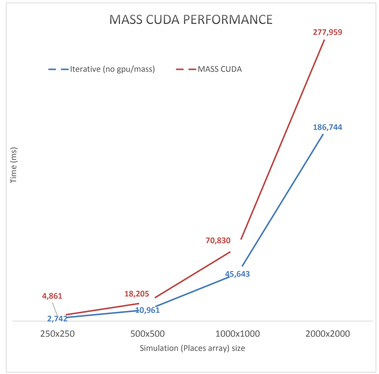

Figure 2: Heat2D- Sequential vs MASS CUDA

```
double r = a * dt / (dd * dd);
//configuration parameters
dim3 dimGrid( (size - 1) / WORK_SIZE + 1, (size - 1) / WORK_SIZE + 1 ),
             dimBlock( WORK_SIZE, WORK_SIZE);
//create host data
double** z = new double*[size];
for (int i = 0; i < size; i++)
{
    z[i] = new double[size];
}
unsigned int nBytes = sizeof(double) * size * size;

//pitch linear input data
double *d_src, *d_dest;
size_t d_pitchBytes;
double* hostar = new double[size * size];
CATCH( cudaMallocPitch((void **) &d_src,
                       &d_pitchBytes,
                       size * sizeof(double),
                       size) );
CATCH( cudaMallocPitch((void **) &d_dest,
                       &d_pitchBytes,
                       size * sizeof(double),
                       size) );

setCold2D<<<dimGrid, dimBlock>>>(d_dest, d_src, size);
CHECK();

int t = 0;
for (; t < max_time; t++)
{
    setEdges2D<<<dimGrid, dimBlock>>>(d_dest, d_src, size, t, heat_time, r);
    CHECK();
    euler2D<<<dimGrid, dimBlock>>>(d_dest, d_src, size, t, heat_time, r);
    CHECK();
    double *swap = d_dest;
    d_dest = d_src;
    d_src = swap;
} // end of simulation
```

Figure 3: Heat2D Test Code Sample

Performance Analysis

Performance of the library was tested through creating an arbitrary program that made use of the functions invoking the GPU, specifically the callAll and exchangeAll functions. These were then compared with the same arbitrary program written in MASS C++. Variables within the tests were changed and recorded, such as computational complexity and combination of the calls.
After the initial tests, a Heat2D program simulating the transfer of heat across a metal material was ran, using various designs and CUDA configurations, to determine how performance improvements can be applied to the library. Figure 3 shows a snippet of version of Heat2D with CUDA.
The following three cases provided insightful information that can be applied for improvements:

- Running Heat2D with primitive array types
- Running Heat2D with Place/State objects without using the library
- Running Heat2D in CUDA with coalesced memory accesses

Proposed Improvements

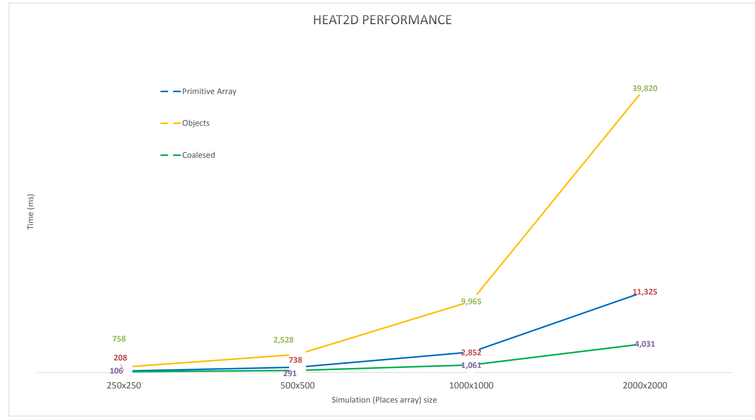

Figure 4: Heat2D performance results compared from the three test types.

Test Type	Description	Library Implementation
Primitive Array	Test ran without using Place/State objects. Much higher performance at the cost of losing programmability (user not defining Place and State objects and instead providing primitive arrays to store data).	Instead of extending Place and State objects, user defines data in primitive arrays beforehand. Possible additional functionality for more advanced users that want to tune performance.
Objects without library	Test ran using Place/State objects without making use of MASS library. CUDA functions explicitly written and called by program. Faster performance when not running through extra library code (partitioning logic, memory transfers from host to GPU)	Refactoring/rewriting library without additional functionality not currently being used (partitioning logic, multiple GPU's). Implementing more important library functions first (callAll, exchangeAll) and verifying performance.
Coalesced Accesses	Primitive type array ran in CUDA kernel functions in native 2D as opposed to 2D array converted into 1D array. Faster memory accesses and increased performance.	Array of Places/States in library use native dimensions (1D, 2D, 3D). Multiple execution flow/functions for different dimensions.

Developer Tools

Poster Template Designed by Genigraphics ©2012
1.800.790.4001 www.genigraphics.com