Multi-GPU MASS Library

Enabling fine-grained parallelism using agent-based programming on GPUs

Rob Jordan, MSCSSE
December 8, 2012

Abstract

The goal of the project is to implement the Multi-Agent Spatial Simulation (MASS) library in CUDA-C using multiple Graphical Processing Units (GPUs) in order to realize a significant speedup over the existing single GPU and multi-threaded versions. MASS is one component of the Sensor Cloud Integration research effort lead by Professor Munehiro Fukuda at the University of Washington Bothell. Though various versions of the MASS library have already been implemented in Java, none of the versions have been able to achieve the level of performance required. We are hopeful that a multi-GPU version can provide the requisite performance.

Introduction

We provide background information on General Purpose Computing on Graphics Processing Units using NVIDIA's CUDA architecture and then describe how this technology can be applied to the MASS library. Next we will discuss the sample application, Wave2D, used throughout this project. Lastly, we will conclude the introduction with a brief survey of related work distilled from an extensive literature survey.

GPGPU with CUDA

Over the last ten years, GPUs have evolved into massively multi-threaded engines capable of performing incredible amounts of parallel computation. Figure 1 shows the raw computation power of GPUs as measured by Theoretical GFLOPs compared to Intel's mainstream CPUs. Researchers have begun harnessing this computational power for tasks other than computer graphics in what is known as general-purpose computing on graphics processing units (GPGPU).

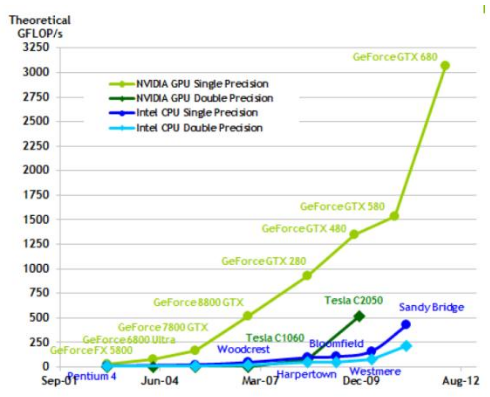


Figure 1 GPU Performance

NVIDIA is a lead manufacturer of GPUs and has developed Compute Unified Device Architecture (CUDA) in order to facilitate GPGPU. Applications are written in C/C++ using CUDA-C extensions, compiled using the NVIDIA compiler, and then executed on a platform containing a CUDA-capable GPU. Before CUDA, developers had to write applications using the graphics-specific processing pipeline, which required using graphics domain concepts like texture shaders for non-graphics programs, which was extremely difficult, time consuming, and error prone.

The general programming pattern for CUDA programs is a master-slave style as illustrated in Figure 2. The CPU acts as the master and the GPU serves as the slave. The CPU initializes the program and copies data from system memory to the GPU memory. The program is executed on the GPU using GPU memory, and then the results are transferred back to the host.

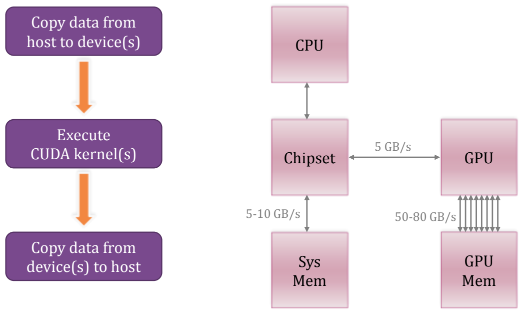


Figure 2 CUDA Programming Pattern

Due to their massively multi-threaded nature, GPUs are exceptionally well-suited for multi-agent simulations and models since, conceptually, each agent or location can be mapped to an independent CUDA thread and executed in parallel. Since GPUs are capable of thousands of parallel threads running on highly tuned ALUs, the bottleneck in programs tend to be bandwidth related. Memory management is key to performance, especially once additional GPUs are added since all communication must use the same PCI-e bus.

Multi-Agent Spatial Simulation (MASS) Library

MASS is a middleware library that provides job parallelization using entity-based programming. The Java specification was written by Professor Fukuda as one component of the Sensor Cloud Integration research effort. Though it is intended to be used for predicting temperatures in Eastern Washington apple orchards in our research project, the library offers general parallelization for any multi-agent spatial simulation model. Figure 3 shows the MASS library being used to model temperature and air flows.

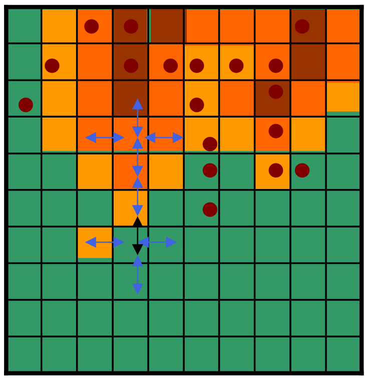


Figure 3 MASS Agents and Places

There are two primary entities in the MASS library: Agents and Places. A Place is an abstract class that represents a fixed location in the simulation space and can exchange data with other Places. Places (a collection of Place entities) is a multi-dimensional array of elements. Places are mapped to threads. Agents are also an abstract class; however, an Agent represents the computational entities that can migrate from Place to Place and interact with other Agents. Different than Places, Agents are mapped to processes.

The MASS library includes an API that enables a developer to write solutions that instruct all Agents or Places to call a certain function, or only a subset of all Agents or Places to call a function. With these tools, a developer can create complex multi-agent simulations that will automatically run in parallel.

To date, only a Java version of the MASS library running on a cluster of nodes has been implemented. A single-GPU version is currently under construction, as well as a multi-node C++ version.

Wave2D

Wave2D is an implementation of Schrodinger's Wave Dissemination model over two-dimensional space. It is a time-stepped model that calculates the wave height for each cell in a finite grid space for a given time period. The height z of a cell located at [i,j] at time t is determined by Schrodinger's wave formula:

```
z[i,j](t) = 2.0 * z[i,j](t-1) – z[i,j](t-2) + c^2 * (dt/dd)^2 *
              ( z[i+1,j](t-1) + z[i-1,j](t-1) + z[i,j+1](t-1)
              + z[i,j-1](t-1) – 4.0 * z[i,j](t-2) )
```

In short, the height of any cell is a function of its previous height and the previous heights of its four cardinal neighbor cells.

It has been selected as a suitable test application for MASS for a number of factors. First, the wave height calculation is relatively simple and straightforward to implement yet offers interesting communication needs, namely the need for each cell to exchange its height with its neighbors. Without this constraint, the parallelization of the algorithm would be trivial. By requiring the exchange of data among cells, the implementations must take into account the transfer of data across threads and processes, paying careful attention to the data decomposition of the problem space. With multiple GPUs connected on the PCI-e bus, this problem involves managing a distributed grid over many separate memory spaces, as illustrated by Figure 4.

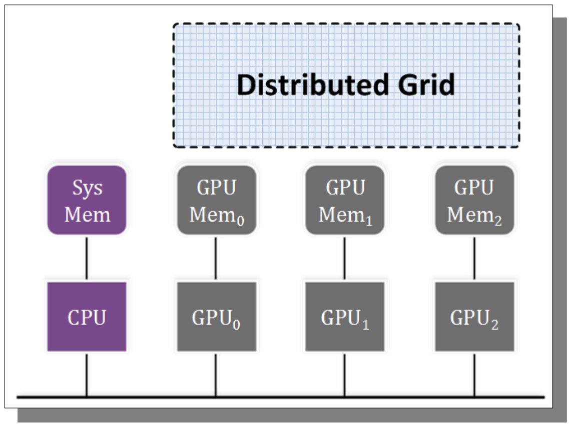


Figure 4 Distributed Grid over GPUs

Secondly, the two-dimensional nature of the application maps perfectly to the intended MASS domain of a spatial model. It also provides an easily adjustable problem set with which to test the scalability of any solution since the simulation space exhibits quadratic growth (i.e., an N x N simulation space). Small simulations can be hand-verified and then the simulation space can be increased dramatically in order to push the system under test to its limits.

Lastly, the intended use of the MASS library within the Sensor Cloud Integration project is to predict temperatures in a wide two-dimensional space using a temperature prediction algorithm not entirely unlike Schrodinger’s wave formula.

Related Work

The literature surveys targeted six aspects of the project: background on the Sensor Grid project and the MASS library, general purpose computing using GPUs, domain decomposition strategies for multi-agent simulations using multiple GPUs, communication versus computation strategies, programming strategies specific to CUDA, and metrics.

Articles [1] and [2] provide background information on the Sensor Grid research project led by Professor Fukuda as well as the motivation behind the Multi-Agent Spatial Simulation (MASS) library. They describe the basic programming style for creating MASS-based applications and how MASS contributes to the overall research effort.

Articles, [3], [4], [5], and [6] provide the technical details of designing general purpose applications using GPUs and CUDA. These articles described the history of GPUs and how they have evolved over the past 10 years from simple graphics processing elements into small-scale super computers in their own right.

These sources also describe CUDA programming and the different types of memory that CUDA exposes through the API. Since memory bandwidth has been shown to be the dominant bottleneck in a CUDA application, proper use of memory is essential to any efficient solution.

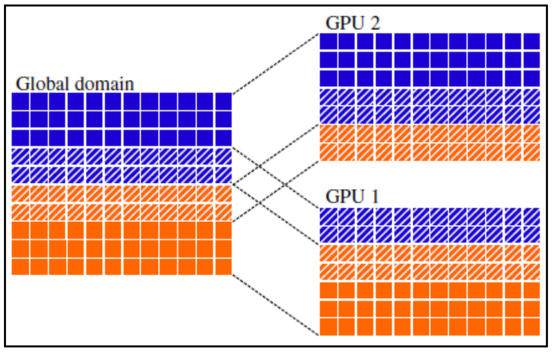


Figure 5 Row domain decomposition with Ghost Cells

Articles [7], [8], and [9], discuss efforts to map multi-agent applications to GPUs with regards to different domain decomposition strategies and how to best map the entities to the GPU threads. The row domain data decomposition strategy described by [7] fit this project's needs precisely so it was selected as the primary method for distributing work across the multiple GPUs. This strategy breaks the domain into stripes based on rows in order to minimize stripe neighbors, thereby minimizing communication, and maps contiguous memory to each GPU so that GPU memory access can be streamlined. Figure 5 demonstrates how the global domain is broken into horizontal stripes using this strategy.

This same article [7] introduced a concept for handling border cells and their need to communicate with neighbors that reside on a different node. Ghost cells (aka. halos and shadow cells in other literature) are redundant copies of data on each GPU. In Figure 5, the Ghost cells are the hashed blue cells on GPU 1 and the hashed orange cells on GPU 2. They are not updated by each GPU but are used for read-only memory access so that the border cells do not need to access memory on a different device. The authors found significant performance improvement using this technique so it was incorporated into this project.

For each GPU,

1. Compute the border cells (Stream 1).
2. Compute the other cells in the grid (Stream 2).
3. Copy bordering cells from GPU to host (Stream 1).
4. Synchronize device (Stream 1).
5. Copy new bordering cells to GPU from host (Stream 1).
6. Synchronize Stream 1 and Stream 2.
7. Repeat.

Figure 6 Overlapping Communication and Computation

The last set of articles, [10] and [11], discussed methods of optimizing a multi-GPU solution by employing overlapping communication and computation. Initial performance timings of Multi-GPU MASS showed that the communication overhead was responsible for the overwhelming majority of the execution time, so a method to reduce this aspect was needed. [11] discussed an automated method of expanding the ghost cell region based on dynamically determined application characteristics. The complexity of their solution did not seem to worth the effort at this stage of the project. The other article [10], however, described a ghost exchange algorithm that fit my application perfectly. In fact, they described my initial implementation attempt and its shortcomings and then presented an optimized solution. The basic algorithm is shown in Figure 6 and utilizes CUDA execution streams to provide simultaneous memory transfer and computation.

Implementation

The implementation strategy had three significant milestones, following an evolutionary development cycle where each successive stage of the process builds upon the previous milestone.

Wave2D served as the test program that evolved throughout this process, transforming from a sequential program executed solely by the CPU to a massively parallel program executed by multiple GPUs coordinated by the CPU.

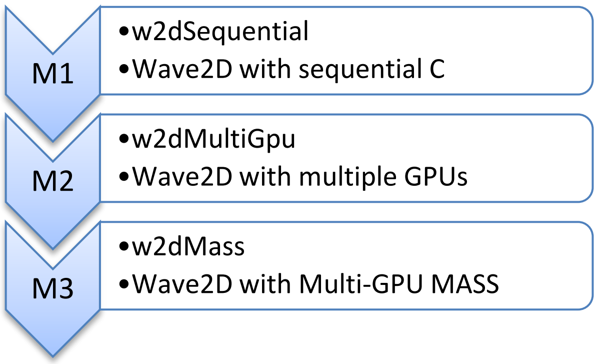
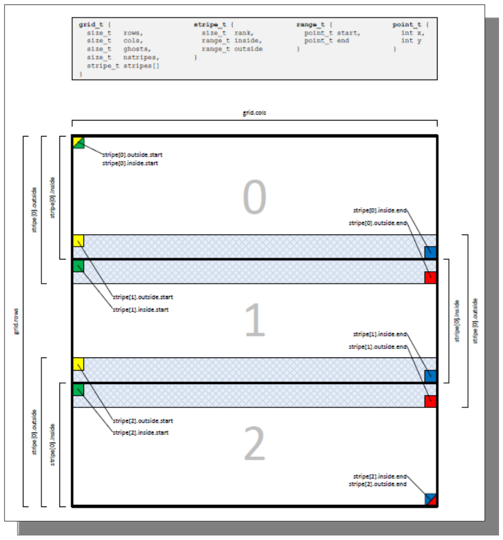

M1
•w2dSequential
•Wave2D with sequential C
M2
•w2dMultiGpu
•Wave2D with multiple GPUs
M3
•w2dMass
•Wave2D with Multi-GPU MASS

Figure 7 Implementaion Milestones

Milestone 1: w2dSequential

Milestone 1 was the baseline implementation of Wave2D in C. This program, w2dSequential, was highly optimized for single-threaded execution and tuned for high performance. It is essentially as fast as possible given the program environment and thus serves as the benchmark against which future milestone implementations are compared.



```
grid_t {                   stripe_t {              range_t {             point_t {
  size_t   rows,             size_t  rank,           point_t start,        int x,
  size_t   cols,             range_t inside,         point_t end           int y
  size_t   ghosts,           range_t outside       }                     }
  size_t   nstripes,       }
  stripe_t stripes[]
}
```

Figure 8 Grid Structure

Milestone 2: w2dMultiGpu

The second milestone implemented Wave2D using multiple GPUs, resulting in the program w2dMultiGpu. Like w2dSequential, w2dMultiGpu is highly optimized for Wave2D and tuned for fast performance. It represents the upper bound of performance for Wave2D on multiple GPUs. This way, we can measure how efficiently the MASS library implementation uses multiple GPUs.

Grid Data Structure

w2dMultiGpu introduced the Grid Data Structure, implementing the row domain data decomposition strategy as discussed in [7]. The structure itself is composed of a hierarchy of inner structures: stripes, ranges and points and enables us to decompose the distributed Places array into stripes across multiple GPUs while preserving the ability to reconstruct global array coordinates.

The grid_t struct contains multiple stripe_t structs that each represent the data for a particular GPU device. The stripes are partitioned such that each stripe is the entire width of the grid. The height of the grid is evenly distributed to each stripe with the exception of any remainder rows.

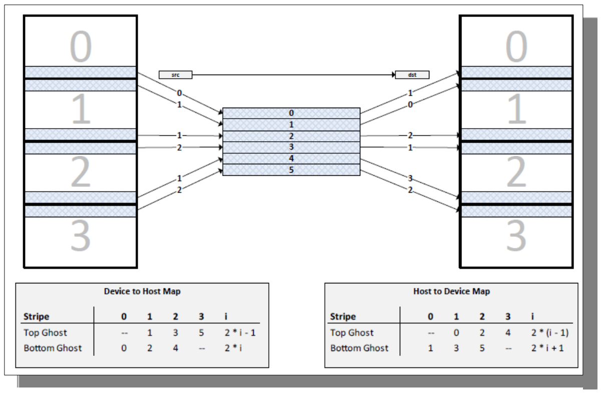
Each stripe contains the ghost data rows from its neighbors. The first stripe only has one ghost region, the overlapping cells from its neighbor below, and similarly the last stripe only has one ghost region from its neighbor above. All other stripes (i.e., the inner stripes) each have a ghost region for the upper and lower neighbor. These regions are represented by the range_t struct. The core data region is also represented by a range.

Lastly, the global locations in the ranges are recorded using the point_t struct, which aids in global and local addressing offset.

The overlapping communication and computation algorithm from [10] and outlined in Figure 6 provided the method for exchanging ghost cells between adjacent stripes.

Ghost Exchange Algorithm

The Ghost Exchange Algorithm, based on [9], uses overlapping communication and computation in order to maximize the GPU compute potential and minimize communication overheads.



Device to Host Map

Stripe	0	1	2	3	i
Top Ghost	--	1	3	5	2 * i - 1
Bottom Ghost	0	2	4	--	2 * i

Host to Device Map

Stripe	0	1	2	3	i
Top Ghost	--	0	2	4	2 * (i - 1)
Bottom Ghost	1	3	5	--	2 * i + 1

Figure 9 Ghost Exchange Algorithm

The lower ghost range from each stripe must be sent to the upper ghost range of the neighbor stripe below, and the upper ghost range must be sent to the lower ghost range of the neighbor above.

This complex interchange of data is facilitated by using the host memory as a buffer where data is copied into the corresponding row and then sent back out to neighboring GPUs.

Since this process is carried out in parallel among multiple GPUs, we must synchronize at various points along the process. Specifically, we synchronize after the copy of data to the host buffer and then synchronize after the copy to the destination GPUs. These two barriers ensure that at the end of the ghost exchange, each GPU contains updated ghost data and can proceed to compute the inner cells.

Milestone 3: w2dMass

The last milestone involved two significant stages: create the Multi-GPU MASS Library and then implement Wave2D using the library. The fundamental principles and lessons learned from the first two milestones were incorporated and refactored into the generalized parallelization library, all the while retaining the row domain data decomposition, ghost cells, and overlapping communication and computation algorithms embedded into w2dMulitGpu.

Methods

Experimental Design

The experiments were performed on the hydra.uwb.edu server in the University of Washington Distributed System Laboratory. This server contains two NVIDIA Tesla C1060 GPUs connected on the PCI-e bus. These GPUs have a 1.3 Compute Capability, 4.0 GP global memory and 30 streaming multiprocessor each. They are theoretically capable of 933.12 GLOPS (single precision).

Each of the milestone implementations were executed for four different simulation sizes (500, 1000, 1500, 2000), on one and on two GPUs, each for 5000 time steps. Each execution configuration was performed 20 times and the results were averaged.

Metrics

The two metrics we focused on are total execution time and speedup. Execution time was measured as simply the time in milliseconds from just after parameter parsing to the end of the simulation. Writing the results to disk was excluded from the execution time.

Speedup represents how much faster the parallel implementation is over the sequential program, and is simply the ratio of sequential execution time to parallel execution time.

Normally discussions involving performance of parallel programs involves the concept of efficiency, which reflects the effect of adding additional threads or processes to a parallel program. Since determining the number of runtime threads on a GPU is virtually impossible due to warp scheduling and thread interleaving, efficiency is excluded from our metrics set.

Results

This section presents the results of the experiments. First the results for w2dMultiGpu are presented, followed by the results for w2dMass.

w2dMultiGpu

The following charts present the execution results for w2dMultiGpu compared to w2dSequential. w2dMultiGpu was executed using a single GPU as well as two GPUs.

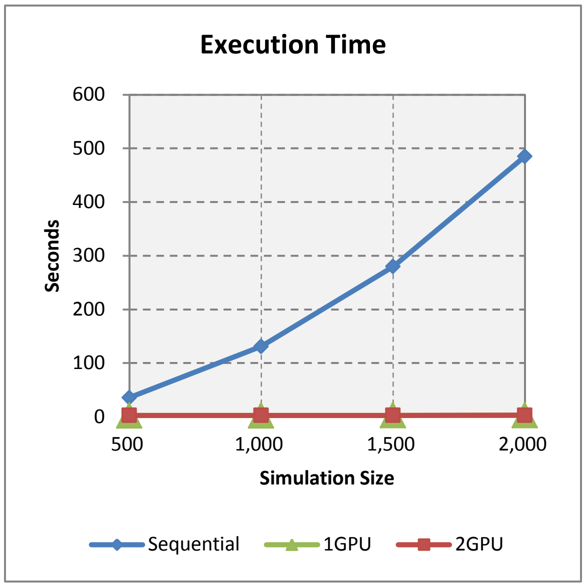


Figure 11 w2dMultiGpu Execution Time

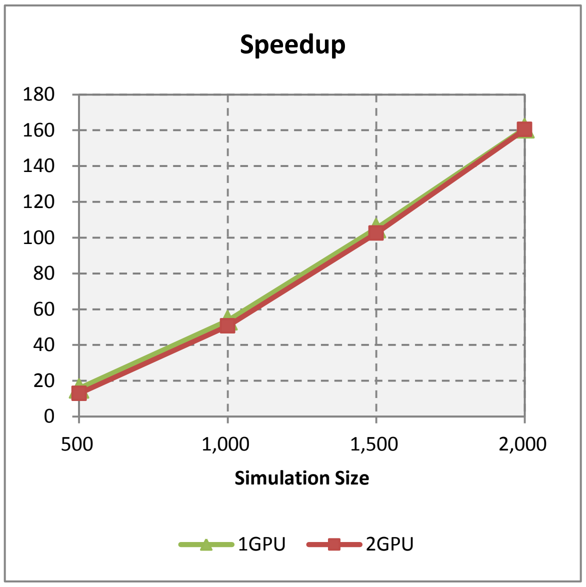


Figure 10 w2dMultiGpu Speedup

w2dMass

The following charts present the execution results for w2dMass compared to w2dSequential. Like w2dMultiGpu, w2dMass was executed using a single GPU as well as two GPUs.

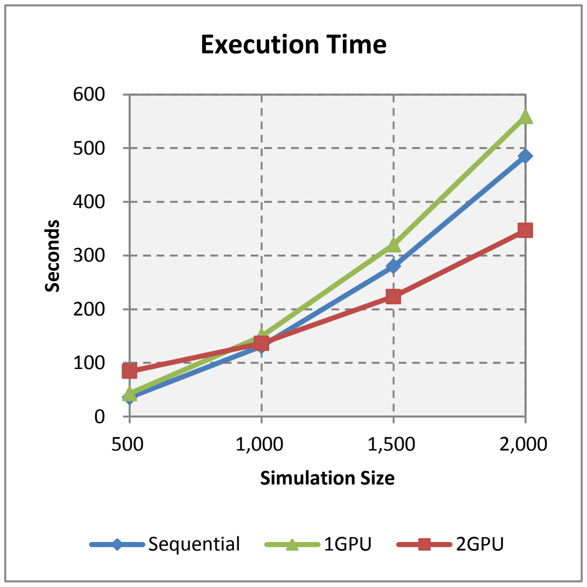


Figure 13 w2dMass Execution Time

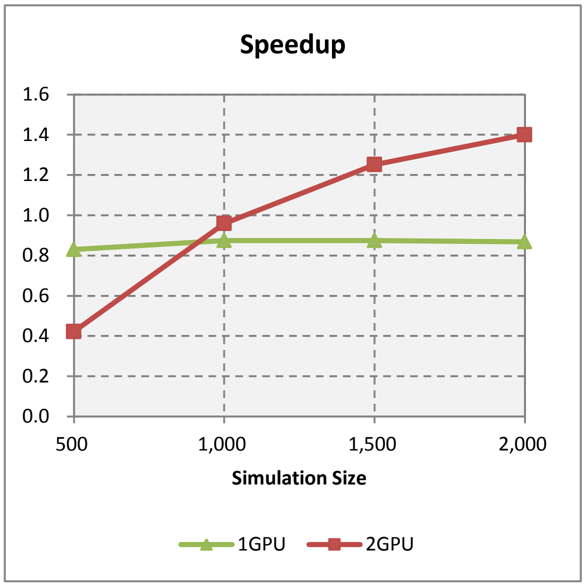


Figure 12 w2dMass Speedup

Discussion

As expected, the performance of w2dSequential degrades as the simulation size increases. This is due to the fact that the domain size grows quadratically yet the program remains sequential. Though the number of elements increases, the time to process each element remains constant which results in the accelerating time to process all elements.

GPUs, on the other hand, have incredible potential for providing significant performance improvements in spatial simulation models as demonstrated by w2dMultiGpu. In stark contrast to the sequential program, the GPU-based Wave2D simulations experienced relatively constant execution time regardless of the simulation size. This constant time behavior is nothing short of incredible in the face of a data set that is growing at a quadratic rate.

As Figure 11 illustrates, the speedup actually increases as the simulation size increases, thereby demonstrating the scalability of the program. This result means that multi-GPU programs are a viable means for performing large-scale spatial simulation computation. Though each GPU has limited memory, we can simply plug more GPUs into a GPU server to scale out horizontally to solve larger and larger problems. The limiting factor is now the host memory since it is used for the ghost exchange and staging ground.

The w2dMultiGpu program validated the two research-based algorithms implemented based on the literature survey. The row domain data decomposition with ghost cells strategy coupled with the overlapping communication and computation algorithm solved the memory bottleneck problem that so often plagues GPU programs.

Refactoring w2dMultiGpu into w2dMass while implementing the MASS library did not provide the level of speedup that was expected. That being said, it is notable that the performance of two GPUs outpaced the single GPU performance once the simulation size reached 1,000 by 1,000. In fact, the multi-GPU solution did outperform the sequential program in all runs with two GPUs and a simulation size over 500. The parallel nature and scalability of GPUs must, at this point, overcome other limiting factors once the simulation size reaches some tipping point.

Comparing the performance results between w2dMultiGpu and w2dMass reveals that though the algorithms remained the same, one or more performance bottlenecks were introduced. One potential bottleneck candidate is known as warp divergence. On the GPU, threads are executed in groups (or warps) of 32. These 32 threads execute the exact same instruction at the same time; consequently, if one thread branches then all other 31 threads must wait until it joins the main execution path again. As such, overall execution time can be reduced by a factor of 31. Boundary conditions are the most common location where these branches can occur and are thus very expensive.

There are two potential solutions to the warp divergence problem. First, the code can be refactored so as to avoid branches and boundaries. While removing these conditions entirely is most likely entirely impossible, any reduction can have a significant impact. A second solution is to group similar operations into similar warps. For example, if boundary conditions cannot be avoided, it would be better to group all threads that will execute along the boundary into the same warp so that there is no divergence. They would all go down the boundary path, leaving the other warps to execute the main code path.

The second potential source for performance bottleneck is the lack of coalesced memory access. On the GPU, memory transactions are per half-warp. This means that that all 16 threads attempt to read memory in parallel. This memory access can be optimized if all threads read adjacent memory addresses. Specifically, these reads can be coalesced if, and only if, the k^{th} thread accesses the k^{th} memory element. This access pattern proves problematic for spatial simulation models since threads tend to access the memory of adjacent locations rather than a single corresponding location. That is, cells often need information from their surrounding neighbor cells and so this coalesced pattern breaks down.

A potential solution to these memory access patterns is to use another type of memory known as texture memory. Texture memory is optimized for exactly the neighbor memory access pattern seen in spatial simulation models and so is an ideal fit for MASS. The problem is that, as of CUDA 4.0, texture memory must be statically allocated, limiting its usefulness in a library that is supposed to provide the flexibility to declare grid sizes at runtime. Additionally, MASS does not limit the developer to a fixed number of Places collections, so the requirement to statically allocate the required memory proves overly restrictive.

A second option to improve memory access times is to use shared memory to cache neighbor elements. Shared memory is located on the device and provides extremely fast access. Furthermore, if is often the case that multiple cells need to access the memory of a particular cell, so we only pay the memory copy price once and then reuse the cached copy. The downside is the complexity of managing this additional layer of memory, ensuring that it does not become stale and avoiding race conditions. Managing these conditions is an area for future research.

Next Steps

Though w2dMultiGpu demonstrated the potential that GPUs have to offer for the MASS library, the implementation of w2dMass failed to reach the target execution speed goals. On the bright side, it appears that w2dMass increases speed as additional GPUs are added to the system. It would be interesting to execute w2dMass on a server with more than two GPUs in order to verify this hypothesis.

The warp divergence and coalesced memory access are other areas for improvement that would benefit from additional research and development. While far from rudimentary, the current w2dMass implementation only tackles high-level memory access patterns. Lower level (i.e., thread- or block-level) optimizations may significantly improve the program's performance.

Another potential area for improvement is the introduction of standard object-oriented programming techniques like abstraction and inheritance to faithfully implement the MASS library specification. As previously mentioned, the Tesla GPU devices in the laboratory have a compute capability of 1.3, which limits the programming techniques available to developers. Devices with compute capability of 2.0 or higher enable the use of abstract classes and function pointers, as well as the linking compilation stage which allows the code files to be separated into header and source files.

Bibliography

[1] M. Fukuda. Agent-based workbench for on-the-fly sensor-data analysis. *Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on:* 333-339, August 2011

[2] J. Emau, T. Chuang, and M. Fukuda; A multi-process library for multi-agent and spatial simulation. *Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on*, 369-375, August 2011

[3] E. Wu and Y. Liu. Emerging technology about GPGPU. *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on:* 618-622, November 2008

[4] NVIDIA. NVIDIA CUDA C Programming Guide. www.nvidia.com. NVIDIA, April 2012.

[5] NVIDIA. NVIDIA CUDA C Best Practices Guide. www.nvidia.com. NVIDIA, July 2012.

[6] J. Sanders and E. Kandrot. *CUDA by Example*. Upper Saddle River, NY: Addison-Wesley, 2011. Print. p. 73–80.

[7] M. Saetra and B. André. Shallow Water Simulations on Multiple GPUs. *Proceedings of the 10th International Conference on Applied Parallel and Scientific Computing* 2: 56-66, 2012.

[8] Y. Liu, K. Shi, H. Deng, and E. Wu. A Multi-GPU Based Semi-Lagrangian Fluid Solver. *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry,* VRCAI '11: 321-26, 2011.

[9] B. Aaby, K. Perumalla, and S. Seal. Efficient Simulation of Agent-based Models on Multi-GPU and Multi-core Clusters. *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques,* SIMUTools '10: 1-10, 2010.

[10] D. Playne and K. Hawick. Asynchronous Communication Schemes for Finite Difference Methods on Multiple GPUs. *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing:* 763-768, May 2010.

[11] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. *Proceedings of the 23rd international conference on Supercomputing*: 256-265, June 2009.

Appendices

Source Code Management

w2dMultiGpu and w2dMass are under source code management on Assembla at https://www.assembla.com/code/multi-gpu-mass/git/nodes. Professor Fukuda is an owner of the project and can grant access to the project.

The project contains six branches as outlined in the following table:

Branch	Description
Master	Multi-GPU MASS Library
mass-sample	A simple program showing how to use the Multi-GPU MASS Library
w2dMass	Wave2D implemented using Multi-GPU MASS, as discussed in this paper
w2dMultiGpu	Wave2D implemented using multiple GPUs (not MASS), as discussed in this paper.
w2dSingleGpu	Wave2D implemented using a single GPU (not MASS). Used as an intermediate development step towards w2dMultiGpu.
w2dSingleTexture	Proof of concept of using GPU texture memory to implement Wave2D. Uses a single GPU.

Assembla uses git, which is an incredibly powerful source control management system. If you don't know git, I highly recommend investing the time to become familiar with it.

Local Source Code

Current versions of the source code can be found in the dslab directory on the uwb file servers. The following table outlines the path to each project. All paths are relative to dslab home (~dslab).

Project	Path
Multi-GPU MASS Library	MASS/multi-gpu-mass/master
w2Mass	MASS/multi-gpu-mass/w2dMass
w2dSequential	SensorGrid/Applications/Wave2D/C/w2dSequential
w2dSingleGpu	SensorGrid/Applications/Wave2D/GPU/w2dSingleGpu
w2dMultiGpu	SensorGrid/Applications/Wave2D/GPU/w2dMultiGpu

Compiling

Each of the GPU solutions require that the NVIDIA CUDA toolkit and developer drivers be installed. CUDA downloads are available here: https://developer.nvidia.com/cuda-downloads.

The systems in the dslab should already have these prerequisites installed. If not, contact the CSS Systems Engineer to have them installed.

w2dMass, w2dMultiGpu, w2dSingleGpu, mass-sample are all make-enabled. That is, simply execute

```
$ make
```

in the project root folder. This command will build two versions of the program: a release version with very limited (if any) debugging information and a debug version that generally provides verbose debugging information. The compiled programs will be placed into the project's `bin/` directory.

If compiling on hydra.uwb.edu, you need to make a slight adjustment to the makefile because the nvcc compiler is installed in a non-standard location on hydra. Comment out (using #) the standard nvcc compiler location and uncomment the hydra nvcc install location.

Execution

CUDA programs can be executed just like any other C/C++ program. No special invocation is required.

Each of the programs take a number of command line arguments. To see the usage, simply execute the program without any parameters. All parameter options will be written to standard out.