Implementing MASS on GPU

1 Introduction

Graphics processing units, GPUs, have evolved (Nickolls & Dally, 2010) from a configurable graphics processor to a programmable parallel processor capable of running graphics applications as well as computational applications. The use of GPUs for general-purpose computation gave rise to the term GPU computing¹, which is the use of GPUs for scientific and engineering computational applications. Utilization of the GPUs for running parallel computational applications would lead to substantial performance improvements as opposed to running on traditional CPU cores in parallel. This is because GPUs are composed of hundreds of cores capable of executing a function on multiple types of data using parallel lightweight threads.

Fig. 1. GPU Computing involves cooperation of a multi-core CPU and hundreds-of-cores GPU used to speed up the computation of a scientific or engineering application by taking advantage of parallelism inherent in the GPU. (Retrieved from http://www.nvidia.com/object/GPU_Computing.html on 12/24/11 1:09 AM PST)

The CPU and the GPU has fundamental differences in their architectures as shown in Fig. 2 below. However, combining the computational power of the GPU with traditional CPUs would result in performance gains for parallel applications by running the parallel portions of code on the GPU and leaving the less compute portions with lower memory bandwidth to run on the CPU.

Fig. 2. GPUs currently have a higher memory bandwidth compared to CPUs. Hence, the GPU can process more floating-point calculations per second and is optimized for execution throughput in terms of executing massive amounts of parallel threads. (Adopted from “Programming massively parallel processors: A hands on approach” by (Kirk & Hwu, 2010))

The sections below describe the enormous computational gains that can be achieved by implementing a parallel library on both the GPU and the CPU using NVIDIA’s CUDA API. In this case the sequential and less bandwidth-intensive portion of code would be run on single-to-multi-core CPUs while the GPU would be used for the parallel portion to improve the overall performance of these library function calls.

2 Background on MASS

Multi agent spatial simulation, MASS, is a library of functions (Emau, Chuang, & Fukuda, 2011) that includes a distributed array of elements known as Places, and multiple simulation entities known as Agents. These high-level abstractions encapsulate simulation such that Place elements can interact with other Place elements as part of a computation, also a single Agent or multiple Agents can reside in a Place, migrate to other Place elements and perform computations. Agents can also perform actions on other Agents as well as interact with Place elements. For example, a game simulation² consisting of territories and military units could map well with a Places acting as the territories and Agents as military units. In this case the military units have the ability to exchange distance information with each other as well as what territories they are currently resident in.

2.1 CallAll()

This function takes as an argument the function to be executed in parallel on all Place elements and/or Agents. Since the Places and Agents are network-independent, they can be spread across multiple clusters of workstations. For example, using the game simulation

analogy, CallAll() can be invoked with a function named update_locations() passed as an argument. The function of update_locations() would be to update the positions of all military units or Agents in the simulation so each agent can know the relative positions of every other agent in to know when movement has occurred and how close agents are to each other.

2.2 ExchangeAll()

The functionality of ExchangeAll() in MASS is to execute a function (in parallel) on a Place element as the source and every other Place elements as the destination of the execution. Arguments passed into ExchangeAll() are: the function to execute, and the indexes of specific Place elements to execute this function on (at most N-1 Place elements, at least N-N+1 Place elements). For example, applying the same game simulation analogy, ExchangeAll() would be called on a single military unit (or Agent) with a function called move_unit() to move this agent to another location, while simultaneously letting every other agent know their new relative position to this “moved” agent.

3 Why CUDA?

Compute unified device architecture, CUDA, is a programming model introduced in 2007 by NVIDIA (Kirk & Hwu, 2010) to support running general purpose scientific and engineering applications on both the CPU and GPU. CUDA-capable GPUs have been aggressively marketed by NVIDIA, and hence have a very large presence in the market place with a substantial customer base (Kirk & Hwu, 2010). And to demonstrate the popularity and/or success of CUDA, by 2010, there have been thousands (Nickolls & Dally, 2010) of applications and technical papers written about CUDA, and this trend is continuing with NVIDIA actively upgrading the GPU device and releasing the latest version\(^3\) of the CUDA API.

4 Implementing CallAll() using CUDA

As seen in Fig. 4 below, CallAll() is implemented in Places as a function that executes the kernel function passed as an argument, set_place_indexes(), updating all indexes of all Places by running the set_place_indexes() function as parallel lightweight GPU threads.

```c
__global__ void set_place_indexes( struct Place* tmp_place )
{
    int tid = blockIdx.x;
    // TODO: adjust tid to spawn as many threads as tmp_place->size
    tmp_place[tid].index = tid;
}

void callAll( struct Place* tmp_place, void (*func)(struct Place*) )
{
    // declare memory for GPU device
    struct Place* dev_place;
    cudaMemcpy( (void**)&dev_place, tmp_place->size * sizeof(struct Place) );
    // initialize device variable with CPU variable
    cudaMemcpy( dev_place, tmp_place, tmp_place->size * sizeof(struct Place), cudaMemcpyHostToDevice );
    // perform computation on GPU device
    set_place_indexes<<<tmp_place->size, 1>>>( dev_place );
    // TODO: replace function name with function pointer call on device.
    cudaMemcpy( tmp_place, dev_place, tmp_place->size * sizeof(struct Place), cudaMemcpyDeviceToHost );
    // free memory allocated on GPU device
    cudaFree( dev_place );
}
```

Fig. 3. Implementing CallAll() on Places
// Testing implementation of places.cu

#include "places.cu"

int main( void )
{
    printf( "Testing places.cu..\n" );

    int new_handle = 0;
    int new_size = MAX_PLACE_SIZE;

    printf( "How many places? " );
    scanf( "%d", &new_size );
    struct Places* places;
    places = (struct Places*) malloc(sizeof(struct Places));
    places->init( new_handle, new_size );

    callAll( places->place, (void (*)(struct Place*)) set_place_indexes );

    // print
    printf( "%s->size = %d\n", places->size );
    for( int i = 0; i < places->size; i++ )
        printf( "places.place[%d].index = %d\n", i, places->place[i].index );

    int tmp = places->size - 1;
    printf( "Last Place: places.place[%d].index = %d\n", tmp, places->place[tmp].index );

    // free memory on CPU
    places->free_places( );
    free( places );

    return 0;
}

Fig. 4. Testing functionality of CallAll() on Places

[dslab@hydra versions]$ nvcc testplaces.cu ; ./a.out
Testing places.cu..
How many places? 13
places->size = 13
places.place[0].index = 0
places.place[1].index = 1
places.place[2].index = 2
places.place[3].index = 3
places.place[4].index = 4
places.place[5].index = 5
places.place[6].index = 6
places.place[7].index = 7
places.place[8].index = 8
places.place[9].index = 9
places.place[10].index = 10
places.place[11].index = 11
places.place[12].index = 12
Last Place: places.place[12].index = 12
[dslab@hydra versions]$
5 Performance Evaluation
In terms of performance, the above CallAll() spawns 1 thread for each place, thereby updating the index for each place simultaneously. For a larger array of places i.e. 500 place elements, each Place element is modified by calling set_place_indexes() and updating each Place index. Therefore, for \( N < \text{Places->size} \), the performance of running the function set_place_indexes() on every place in places is \( O(1) \). However, for a function, sample_function() with a performance of \( O(N) \), the performance of running this function on CUDA with CallAll() for all \( \text{Places} \) will be \( O(N) \).

6 Limitations
Despite the mouth-watering gains that can be achieved by using GPUs to implement the MASS library functions there are some drawbacks mentioned below:

6.1 No support for automatic memory allocation on GPU
MASS functions can be implemented to run on the GPU using GPU APIs like CUDA or OpenCL, however, these APIs require memory to allocated on the GPU for every function that needs to execute on the GPU device beforehand. This poses a huge challenge to implementation of a CPU function by a MASS library user that wants to take advantage of the GPUs power without having to rewrite this CPU function to run on the GPU. This limitation arises because memory is not automatically allocated on the GPU when the GPU API function is called. Automatic memory allocation is supported\(^4\) for C/C++ on the CPU, but APIs CUDA and OpenCL (extensions of C), do not provide support for automatic memory allocation on the GPU. This makes it difficult to write a generic C function that would have a fair chance of executing on the GPU without knowing how much memory this function would use beforehand.

6.2 Client has to know how to program in target GPU API
The limitation above that describes missing support for automatic memory allocation to result in the developer learning how to program in the target API to have the specific function run on the GPU. This might not be such a bad thing considering the minimal learning curve required to perform parallelization on GPU APIs like CUDA. However, this goes against the fundamental design of MASS library which requires users to extend MASS by writing CPU functions that would “invisibly” run on the GPU installed.

7 Proposed solutions

7.1 Implement a compiler (source-to-source translator)
A source-to-source translator (Cooper & Torczon, 2012) is a type of compiler that is dependent on programming languages instead of a specific instruction set of a computer. Implementing a compiler in this case is one possible solution to the problem of a generalized CPU-only function that maps to the GPU in terms of automatic memory allocation on the GPU. Here, the compiler would re-write the code for those functions that are passed as arguments into the MASS library API functions. These functions are translated from CPU executable functions, to GPU executable functions (specifically for CUDA-capable GPUs). This approach would eliminate the need to manually port functions that need to execute on the GPUs and instead be translated by MASS’s custom compiler before being executed.

![Diagram of CPU function as a GPU function](image)

Fig. 6. Executing a CPU function as a GPU function (using CUDA) as multiple parallel threads

As seen from the above diagram, a CPU function is written without any knowledge of whether it would be executed on the CPU or the GPU, and after being translated by MASS’s custom compiler, this function would be capable of executing on the GPU just like any function written to run on the GPU as a lightweight set of parallel threads.

7.2 Educate developer on GPU computing using CUDA
GPU computing is a newly emerging field and due to the fast-growing pace at which new technology is adopted, it would be beneficial for any developer that cares about performance.
Learning to write parallel applications in CUDA has been made easy (Sanders & Kandrot, 2011) in terms of scaling from hundreds of parallel threads to thousands of parallel threads. All the developer has to do is implement the algorithm for the single program that would run on multiple data elements to take advantage of the single program multiple data, SPMD paradigm. In SPMD, the multiple instructions in a program can be executed in parallel on different data elements, in the case of MASS, different Place elements.

8 Conclusions
This independent study investigated the use of GPUs to significantly improve the performance of the MASS library function by introducing a low-level implementation layer on the GPU through parallelism on lightweight GPU threads. Developers implementing applications using MASS library functions can take advantage of GPU parallelism without having to worry about implementing parallel algorithms. The major limitation of the CUDA API and other GPU APIs like OpenCL in implementing GPGPU programs is the inability to implicitly allocate memory on the GPU for client programs that need to run a GPU function. One way to overcome this limitation is to implement a compiler that would convert a CPU function into a GPU function and also parse all the data structures needed for the computation so that memory is allocated before this function is run on the GPU. While this solution to provide a generalizable CPU function that can be translated to a GPU-runnable function is attractive, it would also be beneficial to the developer to gain some knowledge on programming using GPU APIs.

9 References

Cooper, K. D., & Torczon, L. (2012). Engineering a compiler (2nd ed.). Amsterdam ; London: Elsevier/Morgan Kaufmann.

