Parallelizing a Data Intensive Lagrangian Stochastic Particle Model Using Graphics Processing Units

by

Jonathan George Hurst

B.S., Worcester Polytechnic Institute, 2005

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2010

This thesis entitled:
Parallelizing a Data Intensive Lagrangian Stochastic Particle Model Using Graphics Processing Units
written by Jonathan George Hurst
has been approved for the Department of Computer Science

Prof. Henry Tufo

Prof. Manish Vachharajani

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Hurst, Jonathan George (M.S., Computer Science)

Parallelizing a Data Intensive Lagrangian Stochastic Particle Model Using Graphics Processing Units

Thesis directed by Prof. Henry Tufo

Atmospheric transport and dispersion (T&D) models play an important roll in United States national defense. Due to operational time constraints, less sophisticated models have consistently dominated the defense market. Recent advances in graphics processing units (GPUs) and their programming models have made GPUs an attractive platform for commodity, low-power, high-performance parallel computing. Two GPU accelerated (using NVIDIA Corporation's CUDA technology) versions of a sophisticated, large-eddy simulation (LES) based, Lagrangian stochastic model, developed at the National Center for Atmospheric Research (NCAR), were implemented and compared against their single and multiple core CPU (Intel Harpertown) counterparts. The implementation representing the shortest route to GPU acceleration observed a single GPU speedup of 14x over the single core CPU implementation. A more robust and scalable single GPU implementation observed speedups of 20x over the single core CPU implementation.

Dedication

I dedicate this effort to Faith and our blessed life together.

Acknowledgements

It is a pleasure to acknowledge and thank my colleagues at the National Center of Atmospheric Research (NCAR) and STAR Institute who made this effort possible. Dr. Paul Bieringer originally brought this thesis topic to my attention and provided the resources to make it possible through tuition reimbursement at NCAR and fiscal compensation at STAR Institute. Scott Longmore, Ryan Cabell and Ka Yee Wong all helped me to balance my NCAR project demands while bringing this thesis to completion. Dr. Jeffrey Weil and Dr. Jeffrey Copeland instructed me in the science behind Lagrangian particle dispersion, computational fluid dynamics, and the large-eddy simulation based Lagrangian stochastic model used in this effort. John Exby spent a good deal of time and energy as the system administrator of our CUDA analysis workstation at STAR Institute. I thank you all.

A special thanks, also, to my thesis advisor, Prof. Henry Tufo, and my thesis defense committee members, Prof. Jeremy Siek and Prof. Manish Vachharajani.

Contents

Tables

Figures

Chapter 1

Introduction

Atmospheric transport and dispersion (T&D) models play several important rolls in United States national defense. Some T&D models are designed for military or civilian real-time hazard assessment in the event of a chemical, biological, radiological or nuclear (CBRN) agent release [25]. Others are used to generate synthetic environments for designing and testing CBRN sensor technologies and source release characterization algorithms. Due to operational response requirements and constraints relating to available computational hardware, less sophisticated models have consistently dominated the defense market.

Lagrangian particle dispersion models (LPDMs) are a sophisticated and computationally intensive class of T&D model that are designed to simulate realistic particle location tracking through a given meteorological environment. LPDMs can be classified as "embarrassingly parallel" problems because the movement of one particle is independent of all other particles (O(n) computational complexity). The parallelism of LPDMs can, and has, been exploited by high-performance clusters for various applications. However, for applicability to many US national defense related applications, a LPDM model must, at a minimum, perform tractably on the limited resources available in a single high-end desktop or laptop.

In recent years, limitations in semiconductor power and thermal scaling (as outlined by the International Technology Roadmap for Semiconductors) have forced microprocessor developers toward multi-core parallelism. Multi-core central processing units (CPUs), like their single-core parents, excel at executing instructions in minimal time thanks to high clock frequencies, deep instruction pipelines, and instruction level parallelism. For many sequential applications where each processing step cannot execute until the previous processing step has completed, the CPU's approach is satisfactory. Other applications involving data-parallel tasks, such as particle dispersion, stand to benefit from a high throughput, massively parallel, approach where individual instructions take longer to execute, but the total number of instructions processed per unit of time is greater than that of a CPU.

An example of a massively parallel architecture is the Cell Broadband Engine Architecture (CBEA). The CBEA, originally released with the Playstation 3 gaming console in 2005, contains one master core and eight Synergistic Processing Elements (SPEs) that run a reduced instruction set with 128-bit single instruction multiple data (SIMD) vector instructions. The clock rate of the SPEs is comparable to that of a CPU, but the computational throughput of all 8 SPEs each executing two 128-bit vector instruction per clock cycle is much greater. The most recent version of the CBEA, known as the PowerXCell 8i, boasts a single precision floating point (FP) performance of 230.4 GFLOP/s (billion FP operations per second) [8]. The high theoretical performance of the CBEA comes at the expense of application development time and program complexity. The lack of a high level programming model is a significant handicap for adoption of the CBEA.

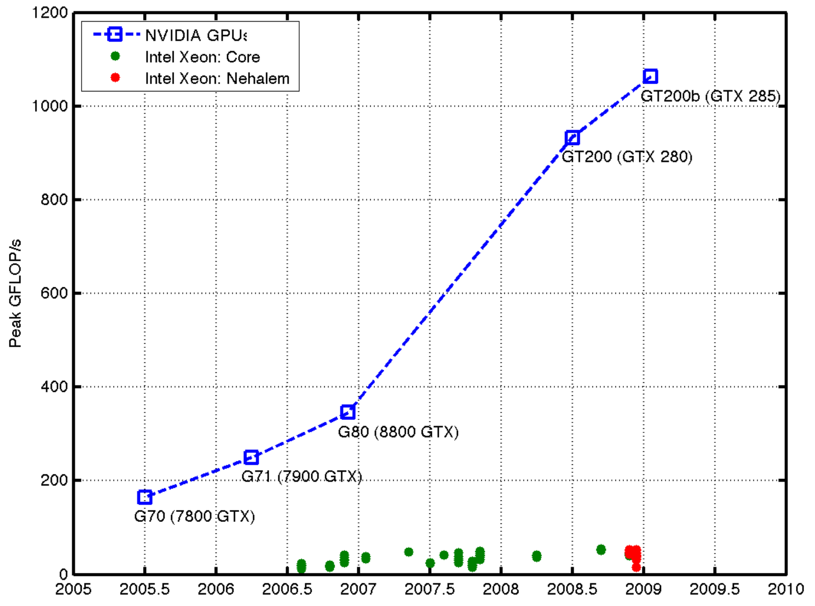
Modern graphics processing units (GPUs) are another example of a massively parallel architecture. Unlike the CBEA which was designed as a general purpose architecture for computation, the GPU was conceived for the sole purpose of accelerating graphics rendering and has evolved slowly, over the past ten years, into a tool for general purpose computation. As a result, the programming models and hardware architectures of GPUs have had more time to hide complicated architecture details and adapt to the needs of the general purpose computing community. GPUs achieve computational performance through hundreds of weak (lower individual computational bandwidth than CPUs) computational cores which share various memory resources. Work is distributed to the cores using a single program multiple data (SPMD) paradigm, where each core runs a small program, called a kernel, on a specific data element. Despite the lower clock rate of GPU cores, the theoretical computational throughput of many NVIDIA Corporation's GPUs (a promising platform for GPU computing) is over an order of magnitude greater than that of CPUs, and the discrepancy continues to grow, as shown in Figure 1.1(a).

GPUs are well suited for acceleration of general purpose applications that resemble graphics calculations with large computational requirements and substantial data-parallelism [15]. Particle models meet both of these criterion. However, one cannot expect to achieve near theoretical computational throughput with every application. The true computational benefit of GPU acceleration on a particle model will depend on how well the model fits into the resources available on a GPU.

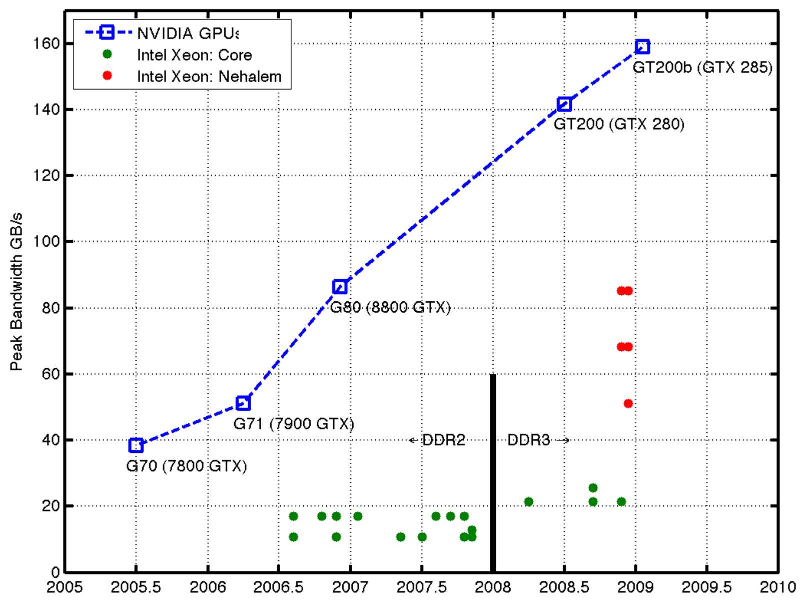
Most GPUs are designed as discrete co-processors attached to a computer through an external bus, or integrated into the motherboard. Memory interactions with a GPU are therefore limited by the internal bus speed of the system. Applications with heavy memory demands will suffer from memory transfer penalties that may reduce the overall utility of GPU acceleration (depending on the ratio of FLOPS per byte of data transferred). However, once data has been copied to the GPU's local memory space, the memory bandwidth between GPU cores and GPU local memory is high, as shown in Figure 1.1(b). Additionally, many GPUs allow applications to perform data transfers to and from a GPU while that device is performing calculations, thus hiding the transfer costs. The balance of memory transfer penalties, improved memory bandwidth on the GPU, and overlapped communication and computation make GPU performance on data intensive applications feasible.

The register and local memory resources of a GPU are limited. A kernel with many variables and copious control structures will likely require more registers than a simple program, and will thus be limited in the number of kernel instances that can occupy the GPU at a given moment. An application with a small register footprint will be better able to occupy a GPU to its fullest potential. The main memory bank of a GPU can have a range of sizes, from 256 MB up to several gigabytes. The memory bank sizes of new GPUs are growing with each release, but it is still quite common for an application to hit a memory wall where computation is impossible. A robust GPU application will need to cope with memory limitations by restructuring the problem domain to fit on the available GPU resource.

This thesis describes the implementation and analysis of two versions of a single memory-



(a) Peak GFLOP/s NVIDIA vs. Intel Xeon



(b) Peak Memory Bandwidth NVIDIA vs. Intel Xeon

Figure 1.1: Peak GFLOP/s and memory bandwidth for a snapshot of the NVIDIA GPU line and the Intel Xeon CPU line. GPU and CPU GFLOP/s were calculated as the product of processor clock speed, number of cores and number of single precision floating point execution units per core. CPU GFLOP/s values were confirmed by the intel export compliance documents [3]. GPU memory bandwidth was calculated as the product of 2 (for DDR memory), the memory clock rate, and the byte width. CPU memory bandwidth was calculated as the product of 2 (for DDR memory), the memory clock rate, the DDR version width (2x for DDR2 and 4x for DDR3), and the byte width. One caveat, Xeon "Core" architectures use the fastest DDR2 or DDR3 memory rate that matched with their front side bus clock rate. Release dates are approximated.

intensive Lagrangian stochastic particle dispersion model using a single NVIDIA GPU enabled with NVIDIA's CUDA developement framework. The first implementation is designed as a "brute force" adaptation of the original FORTRAN based model with minimal code structure changes. The second implementation focuses on decomposing the model's problem-space spatially to achieve scalability to larger problem sizes and to multiple GPUs in the future. We hypothesize that this decomposition will also improve GPU performance by increasing particle locality within the spacial sub-regions of the problem-space.

Chapter 2 will give an introduction to the field of GPU computing and NVIDIA's GPU general purpose computing framework. An introduction to particle models, the large-eddy simulation based Lagrangian stochastic model of interest, and previous efforts to exploit multi-core parallelism in this model will be discussed in Chapter 3. The methods of development and analysis used in this thesis are described in Chapter 4. The brute force translation of the Lagrangian stochastic model to CUDA is described and analyzed in Chapter 5. The scalable CUDA adaptation of the Lagrangian stochastic model, used to cope with large problem sizes and with the potential to utilize multiple GPUs, is analyzed in Chapter 6. The performance of the two CUDA implementations is compared with the original single-core CPU implementation and the multi-core CPU implementation in Chapter 7. Finally, conclusions and future work are covered in Chapters 8 and 9.

Chapter 2

GPU Computing

The GPU emerged in the late 1990s as a way of offloading computationally intensive three dimensional (3D) graphics calculations from the CPU. To meet this need, GPUs were originally designed as fixed function co-processors which would carry a large assortment of geometric primitives through a pipeline of operations to produce an on-screen image. The GPU is called a co-processor because it attaches to the host system through the input/output (IO) system bus. The GPU fixed graphics pipeline later evolved to include small user defined shader programs that would run at each pixel to provide more sophisticated effects[15]. As these programs forced GPU hardware to become more programmable, GPU vendors were actively developing application programmer interfaces (APIs) and libraries to allow general purpose computation on their devices.

While GPUs perform well on the graphics applications they were engineered to accelerate, superior performance is by no means guaranteed for all applications. Applications that exhibit the following characteristics will stand to benefit the most from GPU acceleration:

- **Data-parallelism** The GPU is designed to run a small SPMD kernel at each pixel of a graphics frame buffer. In the case of a graphics application, each pixel has a set of operations that are needed to prepare it for rendering to the screen. This type of parallelism across data elements is referred to as data-parallelism.

- **High ratio of computation to data transfer** Loading data from a GPU's global memory bank is a costly operations on the order of 100x to 300x slower than arithmetic instructions.

A program with more arithmetic work will be better able to hide the latency of individual load and store operations by scheduling arithmetic execution on separate threads while a memory operation is being satisfied. The initial transfer of data to a GPU is also a costly operation relative to FP computation. The ratio of FP operations per byte of data transferred to a GPU is a useful metric in accessing the FP demand of an application.

- **Problem fits on the device** Data transfers to and from the GPU suffer as a result of low memory transfer bandwidth across the system bus. An application that can fit most of its work on the GPU, and keep it there, will suffer fewer transfer penalties.

- **Makes use of device and host** Offloading computation to the GPU leaves the CPU free to perform other tasks while the GPU is occupied.

- **Limited register usage** The fewer registers used by a problem, the more active threads that can be running on the GPU at once. Complex problems that require a large set of registers may be limited in the total performance they can achieve relative to the peak performance of a given GPU.

- **Limited control flow branching** Control flow branching (through **if**, **while**, **for** statements) can cause performance problems when multiple threads travel down different branches of an application. In this case, most GPU architectures would serialize the separate branches of execution, thereby halving the total instruction throughput of the divergent region.

- **Exact numeric precision unnecessary** Graphics calculations generally require only single precision (4 byte) FP arithmetic to provide an accurate solution. As a result, previous generations of GPUs had little or no support for double precision (8 byte) FP arithmetic. This can be a major problem for scientific computing applications which need higher precision arithmetic for accuracy. A simple computational throughput benchmark measured a single precision performance of 693.990 gflops, and a double precision performance of

86.416 gflops on the GPU described in Section 4.1. The latest generation of NVIDIA's GPUs, code named "Fermi", promises a 4.2x speedup in double precision arithmetic performance [14], thus bringing the double precision verses single precision performance ratio from 8:1 on current hardware to approximately 2:1, which is akin to the ratio seen in CPU architectures.

The computational reliability of GPUs has only become a priority with the advent of general purpose computing on GPUs. In the graphics pipeline, hardware issues such as bit-flipping are of minimal impact to the overall task of rendering graphics to a screen. However, scientific applications can be much more sensitive to such computational errors. Sheaffer et al. [24] describes the extent of the problem, and a propose solution. NVIDIA's "Fermi" architecture plans to address reliability using error correcting code (ECC) memory [14].

On older hardware (before "Fermi"), the application which can operate with single precision FP and cope with potential erroneous values will be best suited for GPU acceleration.

This thesis will focus on GPUs manufactured by NVIDIA Corporation and the tools associated with those devices. Specifically, NVIDIA's Compute Unified Device Architecture (CUDA) will be used to implement GPU acceleration into the target application. The remainder of this chapter will describe CUDA in detail, then make mention of the recent standardization efforts of the Open Computing Language (OpenCL). Finally, a sampling of GPU accelerated scientific computing success stories will be references.

2.1 Compute Unified Device Architecture (CUDA)

NVIDIA's general purpose computing framework, tailored specifically for their GPU hardware, is called CUDA. The framework is based around modifications to the C programming language which express GPU parallelism. CUDA provides a compiler to translate code written using the modified C dialect into device binary code to be run on the GPU, while also translating non-device code to be executed on the host CPU. CUDA comes with a high level API called the runtime API, and a low level API called the driver API. In addition to the compiler and APIs, CUDA provides some rudimentary libraries, developer tools, and a set of example applications which explore the features of CUDA. The following sections describe the programming features of CUDA version 2.3, with a focus on the runtime API.

2.1.1 Programming Model

A CUDA function that will run on the GPU is called a kernel. Kernels are written as C functions with the **__global__** qualifier preceding the function specification. Another type of function in CUDA is the device function, labeled by the **__device__** qualifier. Device functions can only be called by kernel functions or other device functions. The contents of device functions will be inlined into the kernel functions at compile time for optimization purposes.

An NVIDIA GPU is organized as a collection of multi-processesors, each containing 8 compute cores. To call a kernel on a device, the host application must specify a series of "blocks" which represent a mapping of the problem domain onto the multi-processors of the device. The size of a block dictates the number of threads to be executed on a multi-processor. Upon invoking a kernel, the device is given a series of blocks to schedule across its multi-processors as they become available. Figure 2.1 illustrates the decomposition of a 2D problem domain into a series of blocks. The quantity of memory resources (registers and local memory) required of each thread can have the affect of limiting the total number of threads that can be assigned to a block; the hardware imposed maximum number of threads per block is 512.

The block of threads given to a multi-processor is processed as chunks of 32 threads, referred to as warps. The threads of a warp are executed in a synchronized fashion with each thread executing the same instruction on the 8 cores of the multi-processor. The 32 warp threads are swapped onto the 8 multi-processor cores in groups of 8 using fast thread switching logic. If any of a warp's threads enter into a different control flow branch of the kernel code, the warp must execute the two separate branches of code sequentially. NVIDIA refers to the execution strategy of warp threads as SIMT (single-instruction, multiple-thread). The SIMT architecture is very similar to SIMD (single-instruction, multiple-data) architectures, except that the software does not need to know the vector width of the hardware. By specifying parallelism as threads, the GPU SIMT architecture hides the hardware details from the user-level software while maintaining computational performance.

The fast thread switching logic also serves to swap warps to and from a multi-processor with no significant overhead. Warp swapping makes it possible to overlap costly global memory operations in one warp, with arithmetic operations in other warps. The more warps contained in a block, the better a CUDA GPU can hide the costs of high latency operations with computational work.

The overhead associated with launching a kernel on a device depends on the number of blocks and arguments given to the kernel. Generally, this overhead is on the scale of multiple micro-seconds to tens of micro-seconds. On the GPU described in Section 4.1, a kernel launch overhead of 12.1 micro-seconds was observed on an empty kernel running with 800 blocks of 128 threads.

2.1.2 Memory Model

CUDA enabled GPUs offer many memory regions with differing levels of locality, latency and bandwidth. Making full use of the low-latency, on-chip memory buffers of the multiprocessors is critical to achieving computational performance, especially in data driven applications where memory bandwidth is the bottleneck. Figure 2.2 gives a visual depiction of the CUDA memory regions. Memory model size and access latency data were derived from the CUDA Programming Guide [13].

- **Global memory:** (Accessible by all blocks. Off-chip.) Global memory, also referred to as device memory, is used to denote the uncached main memory bank associated with the device. It is the most costly memory to access with an access latency of 400 to 600 device clock cycles. In comparison, simple arithmetic operations take only 2 clock cycles. When the threads of a warp are reading from the same contiguous segment of global memory,

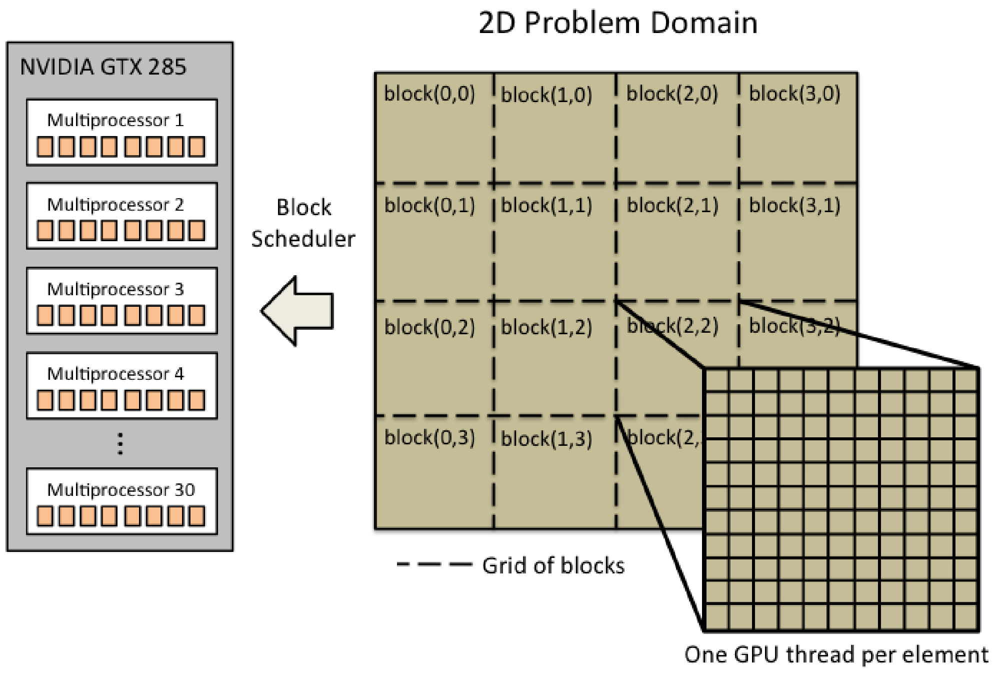


Figure 2.1: Example of a 2D problem domain that has been decomposed into a grid of blocks for execution on an NVIDIA GTX 285 graphics card with 30 multiprocessors. Each block contains a number of data elements which will each be assigned a GPU thread on the device. Blocks are assigned to the GPU multiprocessors by the block scheduler as the multiprocessors become available.

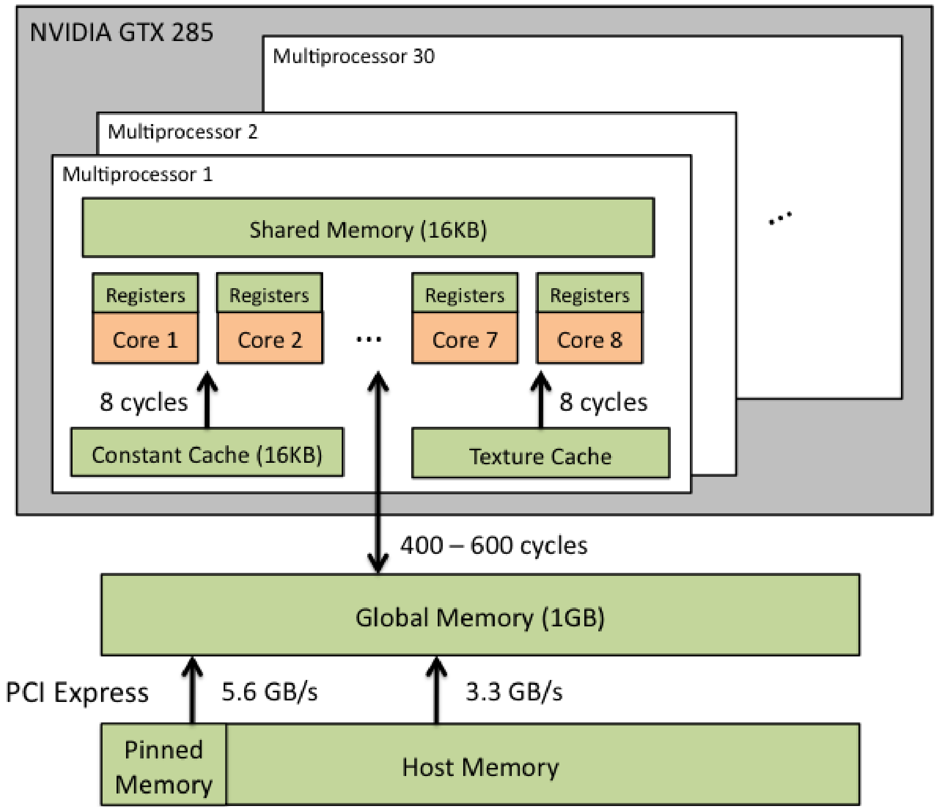


Figure 2.2: Illustration of the CUDA memory model for a GTX 285 device on the host machine described in Section 4.1. Transfer rates over the PCI Express bus were measured using a simple benchmark for pinned, and non-pinned host memory. Clock cycle latencies are derived from the CUDA Programming Guide [13]. Other information is based on the technical specifications of the GTX 285.

the read cost for all the threads is coalesced down into one or two global memory reads. However, if each thread is reading a different area of global memory, each thread will be processed sequentially and will incur its own 400 to 600 cycle penalty. Global memory is allocated from the host using the various cudaMalloc routines, and populated either by a kernel or through a cudaMemcpy command.

- **Texture memory:** (Accessible by all blocks. Off-chip. Cached on-chip.) Texture memory is the same as global memory, except it is read-only and cached based on 2d data locality. The texture cache, located on each multiprocessor, helps to mitigate the performance penalty of non-localized read operations by only reading from global memory when a cache-miss occurs. A cached memory read only costs 8 clock cycles. Unfortunately, if the reads are completely non-localized, not even a cache will help.

- **Contant memory:** (Accessible by all blocks. Off-chip. Cached on-chip.) Constant memory also resides in global memory, and is cached using a special constant cache on each multiprocessor. Cached reads cost 8 clock cycles and uncached reads require a global memory read.

- **Shared memory:** (Local to the block. On-chip.) Shared memory is shared between all the threads running in a block. The amount of shared memory per multiprocessor is 16KB. Access to shared memory is as fast as accessing a register, assuming each thread is accessing a separate element. If multiple threads are accessing the same memory bank, a bank conflict occurs which causes those reads to be serialized.

- **Registers:** (Local to the thread. On-chip) Registers, which have no read or write access penalty, are assigned to a thread at compile time. With only 16 thousand 4 byte registers available per multiprocessor, the number of registers assigned to a thread can have a significant effect on performance. For example, a kernel needing 100 registers per thread is limited to $<=$ 160 threads per block. Unfortunately, there is little a user can do to change this number without making engineering changes (split into multiple kernels) to the kernel code. CUDA does provide a compiler option to suppress the number of registers used by spilling registers to local memory. Register suppression may help with some applications to achieve better performance by increasing their number of threads.

- **Local memory:** (Local to the block. Off-chip) Local memory is a 16KB per block region of global memory that the compiler uses to store structures, static arrays and spilled registers that could not fit in register memory. Because it is global memory, it suffers from the same access penalty. However, the "per thread" structure of local memory means that all local memory accesses from a warp are coalesced.

- **Pinned memory:** (Host memory) The runtime API provides function to allocate and free pinned memory, otherwise known as page-locked memory, on the host machine. Pinned memory has the advantage of increased memory transfer bandwidth between the host and device, and allowing such transfers to be performed asynchronously with kernel executions. For a 4MB host to device memory transfer, the "bandwidthTest" tool the CUDA SDK measure the bandwidths of pinned memory as 5644.7 MB/s and regular pageable memory as 3331.1 MB/s on our analysis machine described in Section 4.1. Pinned memory is a limited resource, however, and allocating too much of it can cause premature memory paging in the host operating system.

2.1.3 Stream Parallelism

CUDA streams offer a means for overlapping memory transfers and kernel operations. By creating multiple streams, the host application can asynchronously trigger memory transfers and kernel invocations while ensuring that invocations on the same stream are executed in order. The CUDA device driver provides this functionality via queues used to store the memory transfers and kernel operations waiting for the device. CUDA also provides query routines to determine the state of the streams at a given moment and to retrieve any error signals. Asynchronous memory transfers have the additional requirement that the host memory associated with the transfer be declared in pinned memory.

2.1.4 Developer Tools

Developing for a CUDA GPU can be quite difficult. In particular, it is often time consuming to diagnose what is happening on the device. CUDA continues to engineer new developer tools for the various platforms used in GPU computing. Some of the tools used on the Linux operating system for this thesis are the following:

- **Emulation mode** The **–device-emulation** compiler flag tells the compiler to translate any kernel code into host code to run on the CPU. The resulting program executes the same code, but using only the CPU as an emulator of the GPU, thus allowing print statements. The emulation functionality is valuable when attempting to diagnose a problem deep inside the kernel. Using the emulation executable with a memory analysis tool like Valgrind (for Linux) can help to find errors related to improper memory access.
- **CUDA Profiler** The CUDA profiler is a helpful way of extracting detailed timing information about what a kernel is doing. An executable instrumented for profiling will generate a report file with timing information, organized to the users specification. The CUDA profiler also allows a user to instrument their code with instruction counters and memory access counters. Unfortunately, the CUDA 2.3 profiler does not have cache hit/miss counters.
- **Occupancy Calculator** The CUDA compiler can provide various details about how a kernel was compiled including, the number of registers used per thread, the amount of local memory used per block and the amount of statically allocated shared memory per block. This information can then be interpreted to adjust the code and determine how many threads should be assigned to each block. CUDA provides an Occupancy Calculator spreadsheet which helps by translating the compiler kernel resource utilization information into multiprocessor occupancy statistics. Multiprocessor occupancy represents the ratio of

active warps per multiprocessor to the maximum number of active warps per multiprocessor. This tool makes it easier for a user to explore the optimization space of the GPU and avoid resource limitations.

2.2 Open Computing Language (OpenCL)

OpenCL is a computing standard produced by the Kronos Group [5] in association with most of the major players in CPU and GPU computing. OpenCL provides the specifications for a C language extension for heterogeneous computing across multi-core CPUs and GPUs. The open standard makes it possible to write an application once, and run it across GPUs from different vendors, so long as they have implemented an OpenCL compiler for their hardware. This is a major advantage for shared applications with diverse user bases. OpenCL comes with the disadvantage that it is more complicated than CUDA as it was designed to operate on a broad range of hardware. OpenCL was still in its inception phase at the outset of this thesis, and thus was not utilized.

2.3 Performance in Scientific Computing

GPUs are already beeing used to accelerate many scientific applications with varying levels of success. NVIDIA's website provides a list of GPU success stories from independent developers, with the most extraordinary speedups in the range of 2600x (genetic algorithm [20]) to 675x (stochastic differential equations [4]) over sequential CPU performance. Ryoo et al. [22] used a CUDA GPU to accelerate several different benchmark applications, achieving modest speedups (over sequential CPU performance) for most applications (10x - 30x), and exceptional speedups for three application (102x - 457x). These published results are at the top the the performance spectrum because they satisfy many of the GPU acceleration preferential characteristics described earlier.

Recent rumors and news reports (no formal publication) suggest that Professor Takayuki Aoki of the Tokyo Institute of Technology and his associates have used CUDA to accelerate an entire version of the Weather Research Forecast (WRF) model for operational use [27]. Until this announcement, the majority of work in the WRF model has related to accelerating individual components [10]. The movement toward GPU acceleration in community models like WRF, with a broad base of users, further reinforces the legitimacy of the GPU movement. It also introduces new issues of code maintainability when CPU and GPU software need to be kept in agreement; a topic of ongoing research.

Chapter 3

Previous Work

3.1 Particle Dispersion and Fluid Dynamics Models on GPUs

Transport and dispersion (T&D) models are designed to model the release of CBRN agents in some realistic meteorologic and geographic scenario. In the event of a CBRN agent release, some T&D models are designed to assist first responders in understanding the scope of the event. Other models are designed for research and diagnostic purposes. T&D models and computational fluid dynamics (CFD) models present an attractive area for GPU acceleration, not only because of the real-time performance requirements of some models, but also because of the structure of their mathematical calculations. Specifically, the data dependencies in these models are well localized.

Prior to the release of high level programming frameworks like CUDA, innovative developers were using the GPU pipeline directly to accelerate T&D and CFD problems. Harris implemented the Navier-Stokes equation for incompressible flows using the GPU pipeline directly [2]. An application speedup of up to 6x was achieved over the CPU version of an elementary CFD problems. Li et al. [7] implemented the Lattice-Boltzman method for CFD using the stages of the GPU pipeline to achieve 15x speedup relative to a sequential CPU implementation. In the T&D field, Willemsen et al.[29] implemented a simple Lagrangian dispersion model and achieved up to 3 orders of magnitude speedup over the CPU based model. Pardyjak et al. [17] extended the Willemsen work to include particle reflection off of building, for urban simulations. They achieved a 180x speedup over a CPU version of the same building aware model.

With the release of CUDA and other frameworks, further work has been done. Senocak et al.[23] created a multiple-GPU implementation of the Navier-Stokes CFD solver achieving up 100x speedup over a CPU implementation using 4 NVIDIA GPUs at once. Molnar et al.[11] implemented a single NVIDIA GPU stochastic Lagrangian particle model for air pollution dispersion which achieved 80-120x speedup compared with a single threaded CPU implementation. Goedel et al. [1] implemented a CUDA version of a Discontinuous Galerkin Finite Element Methods (DG-FEM) which achieved a maximum speedup of 71x over a quad-core CPU implementation.

Moving away from GPU enabled models, Roberti et al. [21] implemented a parallel MPI based Lagrangian stochastic model using CPU compute nodes. Particle movement was driven by a single empirically generated wind field which was distributed to all processors. The problem was parallelized by dynamically splitting the particles among processors to ensure load balancing. The only communication after the beginning of the model is to record the location of particles. Overall, they achieved a 9.3x speedup when running on 10 CPUs.

3.2 Large-Eddy Simulation Based Lagrangian Particle Dispersion Model

The particle dispersion model used in this effort is a Lagrangian stochastic model (LSM) driven by large-eddy simulation (LES) meteorology data[28], henceforth referred to as LES-LSM. In Lagrangian models, the dispersion fields are calculated by releasing passive “particles” into the scenario and tracking them through time. In its current version, the LES-LSM does not account for buildings or terrain features. The LES-LSM is called stochastic because it perturbs the particle velocity by small amounts to achieve realistic spread and flow. Thomson [26] first formalized LSMs to parameterize the stochastic perturbations using a mean Eulerian wind velocity probability density function (PDF), and the wind variance. The LES based model adapts Thomson’s stochastic calculation to utilize the resolved wind velocity and the variance of the sub-grid scale (SGS) velocities provided by the LES.

Large-eddy simulations are a class of model that are able to accurately represent the atmospheric turbulence in the Planetary Boundary Layer (PBL) which is very important for realistic simulations. LES are more realistic because they compute turbulence at the micro beta and gamma scales and because they compute SGS components such as turbulent kinetic energy, momentum fluxes and heat fluxes. A model called the EULerian semi-LAGrangian research model for geophysical flows (EULAG) [18] was used to generate the LES data for this thesis. EULAG output wind and turbulent energy are written as 3 dimensional (3D) gridded binary files containing double precision values at a regular time interval. Other turbulence vertical profiles are written as separate binary files containing double precision FP values.

The LES-LSM model is unique not only because it utilizes more realistic LES class data, but because it uses the SGS information to better represent the small scale turbulence. The equation for calculating the wind vector for particle displacement in the LES-LSM is

$$\vec{u_L}(\vec{x_{os}}, t) = \vec{u_r}[\vec{x_p}(\vec{x_{os}}, t), t] + \vec{u_s}[\vec{x_p}(\vec{x_{os}}, t), t] \tag{3.1}$$

where $\vec{u_L}(\vec{x_{os}}, t)$ is the velocity vector at time t for the particle released at $\vec{x_{os}}$, $\vec{x_p}(\vec{x_{os}}, t)$ is the particle location at time t of the particle released at $\vec{x_{os}}$, $\vec{u_r}$ is the resolved velocity at $\vec{x_p}$ and time t, and $\vec{u_s}$ is the SGS stochastic velocity at $\vec{x_p}$ and time t. The resolved velocity is calculated directly from the 4D gridded LES data through a bilinear interpolation across the x,y,z and time dimensions. The SGS velocity is calculated from a combination of the turbulent energy extracted directly from the 4D LES data, the turbulence vertical LES profiles, and several random numbers provided by a pseudo random number generator.

The location of a particle is updated using the equation

$$\vec{x_p}(\vec{x_{os}}, t + \delta t) = \vec{x_p}(\vec{x_{os}}, t) + \vec{u_L}(\vec{x_{os}}, t) * \delta t \tag{3.2}$$

where δt is the model time-step, which is parameterized to take smaller steps when the SGS turbulence is greater. The locations of the particles are saved as output at a certain frequency denoted by δt_{save}. When an update of $\vec{x_p}(\vec{x_{os}}, t + \delta t)$ makes the $t + \delta t$ time greater than or equal to the next write time, t_{save}, a linear interpolation is done between $\vec{x_p}(\vec{x_{os}}, t)$ and $\vec{x_p}(\vec{x_{os}}, t + \delta t)$ to acquire the location $\vec{x_p}(\vec{x_{os}}, t_{save})$ which is then saved in an output field. The value of the t_{save} is updated to $t_{save} = t_{save} + \delta t_{save}$.

When particles are moved off of the grid along the X and Y lateral dimensions, instead of ceasing to exist, they are placed back on the grid on the opposite side of the dimension that they crossed. For example, a particle surpassing the high bound of the X axis would reappear at the lower bound of the X axis with the same velocity and Y coordinate location as it had prior to the transformation. In the vertical Z dimension, particles that traverse the lower or upper bound reflect back into the volume and the particles velocity values are adjusted to match the reflection. Particles in the lowest level of the LES volume are subject to special treatment to cope with the science of near surface turbulence. The various special cases required to move particles translated directly into control flow branches, which do not translate well to a GPU architecture.

The software written for Weil's 2004 paper[28] to implement LES-LSM was the starting point for the effort described in this thesis. Figure 3.1 shows the processing structure of the original LES-LSM model. The code was written using the FORTRAN 90 programming language as a proof of concept. Double precision FP numbers were used for all arithmetic and data elements. The default configuration of the model simulates 20 temporal particle releases, each injecting a fixed number of particles at 81 evenly distributed spacial locations, specified as a 9x9 grid along the surface layer of the LES data. Distributing the release locations across the domain means the model output represents an ensemble average of the various releases. The model can also be configured to inject particles from a single location. The ensemble average release scenario and single realization scenario are illustrated in Figure 3.2. As output, the model tracks the particle locations relative to their release points to produce a 2D cross-wind integrated concentration grid illustrating the ensemble down-wind dispersion of the particles, and a 3D gridded field showing the ensemble distribution of particles across the x,y and z dimensions. Output is only written to disk at the very end of the simulation. The time steps of the model, δt, has a max value of 0.1 seconds and particle binning time, δt_{save}, is set to 0.25 seconds.

After some discussion with Dr. Weil, we hypothesized that single precision arithmetic would provide sufficient accuracy for the computation performed in this model. Section 4.6 describes the

Algorithm 1 Sequential LES-LSM Details

```
Require: Range of LES files
Require: Particle release schedule (release times, release locations, particles per release)
 1: Read and preprocess first LES file valid at $T_{L0}$
 2: for all Other LES files do
 3:   Read and preprocess current LES file valid at time $T_{L1}$
 4:   Calculate the temporal releases with particles between $T_{L0}$ and $T_{L1}$.
 5:   Initialize $t_{part}$
 6:   for $Rel_{begin}$ to $Rel_{end}$ do
 7:     while $t_{part} < T_{L1}$ do
 8:       for all Release locations do
 9:         for all Particles released at this location do
10:           Get the $t_{part}$ for this particle
11:           while $t_{part} < T_{L1}$ and $t_{part} < t_{save}$ do
12:             Calculate resolved particle velocity at $x_{part}$ and $t_{part}$.
13:             Calculate SGS stochastic velocity at $x_{part}$ and $t_{part}$.
14:             Update $x_{part}$ and $t_{part}$
15:             Save $x_{part}$ to output buffer if $t_{part} >= t_{save}$
16:           end while
17:         end for
18:       end for
19:     end while
20:   end for
21:   $T_{L0} = T_{L1}$
22:   Replace first LES data with current LES data.
23: end for
24: Write model output.
```

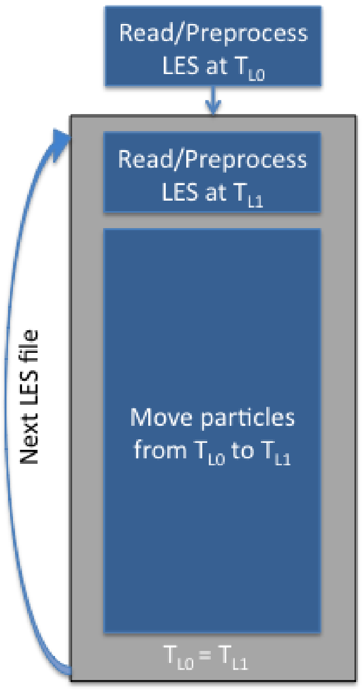
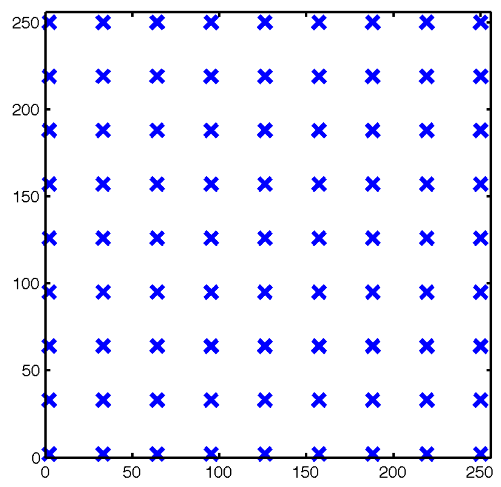
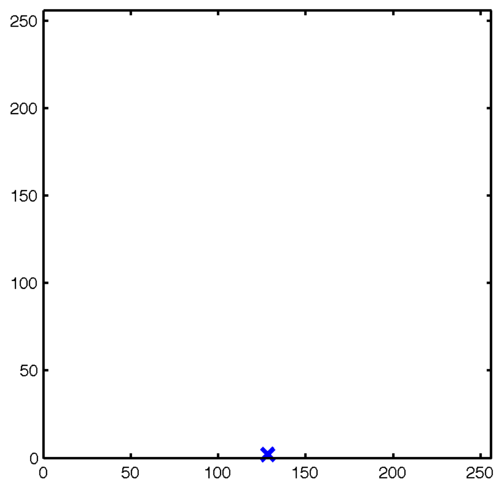

Read/Preprocess LES at T_L0
Read/Preprocess LES at T_L1
Move particles from T_L0 to T_L1
Next LES file
T_L0 = T_L1

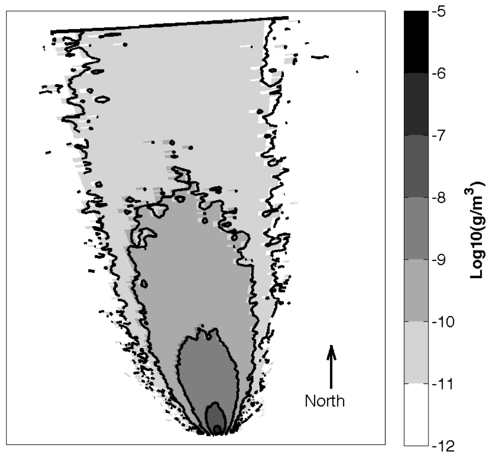
Figure 3.1: A detailed description of the computational workflow of the sequential LES-LSM, coupled with a simplified diagram which highlights the primary computational factors of the LES-LSM represented as blue boxes (primary factors are "Read/Preprocess LES" and "Move particles from T_{L0} to T_{L1}").



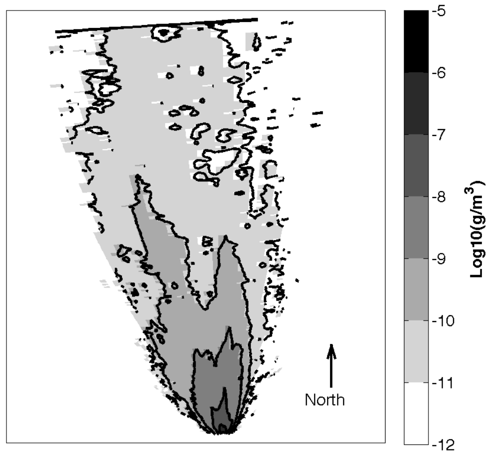
(a) Average Scenario Release Locations



(b) Single Realization Release Locations



(c) Average Scenario



(d) Single Realization

Figure 3.2: Visual depiction of the average release scenario with 81 releases spaced evenly over the domain 3.2(a) and the single release location scenario 3.2(b). Figures 3.2(c) shows the ensemble particle dispersion results of an average release scenario with 81 release locations and 20 release times. Figures 3.2(d) shows the ensemble particle dispersion results of an single release scenario with one release location and 20 release times. The single release scenario produces a more variable field because it represents the winds at a more localized area, while the average scenario produces more of an idealized Gaussian plume.

confirmation of this hypothesis, accomplished by tracking individual particles in the single precision and double precision LES-LSM and comparing them for discrepancies. The use of single precision arithmetic makes it possible for the GPU implementation to achieve maximal performance on our GPU device (due to the discrepancy between double and single precision arithmetic on GPUs). For the sake of simplicity, all performance results shown in this thesis were measured using single precision FP versions of the LES-LSM.

3.3 LES-LSM Floating Point Complexity

The number of FP instructions executed in moving particles between each adjacent pair of LES files was observed using the “Pin” framework for dynamic binary instrumentation [9]. The custom Pin tool, created specifically to observe the LES-LSM, keeps a tally of the number of FP instructions and all other instructions executed in each function contained in, and linked to, the LES-LSM binary. A summary of the previous tallies is written to a log file when each LES file is read. The result of the Pin tool is a time series of log files containing the total number of FP instructions and all other instructions executed inside of each function in the simulated span between two LES files. Figure 3.3 shows the observed sum of all particle movement FP instructions per LES file in the sequential LES-LSM (single precision), running on two of average workloads.

The peaks in each line of Figure 3.3 represent when particles are added to the simulation at the surface of the LES domain and need to be processed using the near surface turbulence calculations (more FP operations). The mean number of FP instructions required in moving one particle between a pair of LES files is 145,127 instructions.

3.4 LES-LSM Sequential Performance

Particle dispersion is considered an embarrassingly parallel problem because the calculations involved in moving a particle are independent of other particles (O(n) computational complexity). The LES-LSM is no exception. However, LES-LSM is more complex of a problem to parallelize because of its data dependencies and the inevitable overheads associated with distributing that data to multiple computational units. Table 3.1 shows a collection of sequential LES-LSM (single precision) performance results generated using the realistic workloads described in Section 4.3.

		Workloads					
		ave1	ave2	ave3	sin1	sin2	sin3
Workload description	Total particles	162000	810000	1620000	162000	810000	1620000
Total time per operation	LES read	1031s	1041s	1039s	1033s	1032s	1034s
	LES preprocess	34s	34s	34s	34s	34s	34s
	Move particles	5208s	24072s	47271s	4274s	21838s	42857s
	Miscellaneous	1s	1s	1s	1s	1s	1s
	Total time	6274s	25148s	48344s	5341s	22905s	43927s
Percent of total time	LES read	16.44%	4.14%	2.15%	19.34%	4.50%	2.35%
	LES preprocess	0.54%	0.13%	0.07%	0.63%	0.15%	0.08%
	Move particles	83.00%	95.72%	97.78%	80.02%	95.34%	97.57%
	Miscellaneous	0.02%	0.00%	0.00%	0.01%	0.00%	0.00%
Amdahl's Law	Theoretical max speedup	5.88x	23.38x	45.05x	5.00x	21.47x	41.07x
FP statistics	Mean FP ops per byte	< 55.0	< 274.9	< 549.7	< 55.0	< 274.9	< 549.7
	Mean GFLOPS	0.452	0.489	0.497	0.550	0.541	0.551
	Mean GFLOPS % of peak	3.765%	4.078%	4.144%	4.586%	4.509%	4.589%

Table 3.1: Results from the sequential LES-LSM (single precision) across the realistic workloads described in Section 4.3. Metrics described in detail in Section 4.2. The Mean GFLOPs % of peak metric uses the single core CPU peak of 12 GFLOPs.

One way of interpreting sequential algorithm performance results is by decomposing the algorithm into sequential sections, which cannot be parallelized, and parallelizable components. This is the logic behind Amdahl's Law [16], which uses the percentage of total application time spent on parallel tasks to derive a maximum theoretical speedup through the equation $S = \frac{1}{1-r}$, where r is the parallelizable fraction of the application performance time. This simple model is applied to the performance results in Table 3.1 using the "Move particles" time as the parallelizable fraction, and treating the LES read/preprocess operations as sequential (in reality, they are not entirely sequential). This simple model of theoretical speedup shows that the LES-LSM will benefit most from parallelization of the particle movement section when larger workloads are used. The mean FP operations per byte statistic shown in Table 3.1 also confirms that larger workloads are much better suited for parallelization.

A model of the sequential LES-LSM was created to help understand where computational time is spent. Model results were measured by running the sequential LES-LSM across the realistic workloads. Figure 3.4 shows the measured performance model parameters with invariant parameters represented by a single mean time values, and variable parameters represented as a linear model on the number of particles, $N_{particles}$. Table 3.4(b) shows the observed sequential LES-LSM performance times and the model predicted times.

3.5 LES-LSM Multi-core Parallelism

The LES-LSM was modified prior to this effort to use multiple CPU cores of a given computer through the Message Passing Interface (MPI) library standard and through the Open Multi-Processor (OpenMP) language extension. MPI provides an open standard for writing distributed memory, parallel applications where users must explicitly manage memory and communication between processes. OpenMP focuses on shared memory parallelism by providing simple constructs to describing threaded parallelism in a sequential application. In both of the multi-core versions, the most simple approach was taken in exploiting parallelism and no application restructuring was done. The two multi-core versions are independent of one another, meaning the MPI version doesn't make use of OpenMP primates, and vice versa.

The MPI multi-core algorithm is illustrated in Figure 3.5. The MPI version is implemented to read the LES data on process 0, and broadcast the entirety of that date to each other process. This is a poor implementation and leads to N copies of the LES data-sets in memory, where N is the number of processes created by MPI. The parallel work for each process comes from the inner-most **for** loop which transports all the particles releases at a given location. This loop is split across all N processes as evenly as possible. No dynamic load balancing is done. Finally, an MPI reduce collects the localized output grids from each process and sums them on process 0 where output is written.

The OpenMP algorithm is illustrated in Figure 3.6. Because OpenMP is a shared memory tool, the pseudo-code for this version is nearly identical to the original. Before the **forall** loop over release locations, OpenMP spawns N threads, and those threads are used to divvy up the work from the inner-most **for** loop. Because shared memory is being used, the save action must be protected from having multiple threads writing to the same location at once. This is done using an OpenMP locking "critical" primitive. After the **forall** loop over release locations, the N threads are destroyed and the main process continues.

The performance of the MPI and OpenMP implementations is shown in Figure 3.7 using a range of 2 to 8 of the 8 CPU cores on the analysis system described in the methodology. Both versions suffer from parallelism overhead which, in combination with the sequential file reads, prevent them from achieving near linear speedup with the addition of more cores. The MPI version performs better than the OpenMP across all the core counts, and the 8-core MPI version will be compared against the GPU implementations in later analyses.

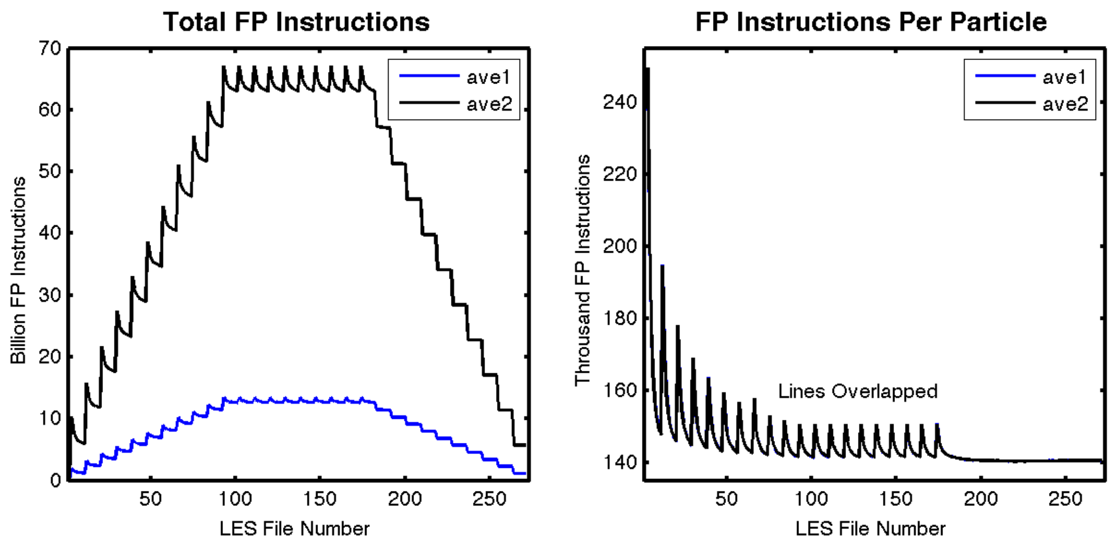


Figure 3.3: The FP intensity of the sequential LES-LSM (single precision), observed using the a custom Pin[9] which tallied the number of FP instructions executed in moving particles between each adjacent pair of LES files.

Operation	Time Mean (sec)	Time Variance (sec^2)	# Samples
Per LES: read	3.8306	0.0756	1080
Per LES: preprocess	0.1254	$9.8133 * 10^{-9}$	1080
	Linear Model		Bandwidth
particle movement	$2.8603 * 10^{-5}(sec/part) * N_{particles} + 0.0344(sec)$		34,960 (part/sec)

(a) Sequential FORTRAN LES-LSM Performance Model Parameters

Workload:	ave1	ave2	ave3	sin1	sin2	sin3
Observed (sec):	6254	25041	47920	5354	22420	43815
Predicted (sec):	5910	22778	43862	5910	22778	43862

(b) Sequential FORTRAN LES-LSM Error

Figure 3.4: Sequential FORTRAN LES-LSM (single precision) Performance Model Parameters and Verification

Algorithm 2 MPI Multi-core LES-LSM

```
Require: Range of LES files
Require: Particle release schedule (release times, release locations, particles per release)
 1: MPI: Spawn $N$ processes
 2: MPI: Assign each process an id: $P_i dx$
 3: MPI: Read and preprocess first LES file valid at $T_{L0}$ if $P_i dx == 0$
 4: MPI: Broadcast first LES to all $N$ processes
 5: for all Other LES files do
 6:     MPI: Read and preprocess current LES file valid at time $T_{L1}$ if $P_i dx == 0$
 7:     MPI: Broadcast first LES to all $N$ processes
 8:     Calculate the temporal releases with particles between $T_{L0}$ and $T_{L1}$.
 9:     Initialize $t_{part}$
10:     for $Rel_{begin}$ to $Rel_{end}$ do
11:         while $t_{part} < T_{L1}$ do
12:             for all Release locations do
13:                 for MPI: $P_i dx$ particle subset do
14:                     Get the $t_{part}$ for this particle
15:                     while $t_{part} < T_{L1}$ and $t_{part} < t_{save}$ do
16:                         Calculate resolved particle velocity at $x_{part}$ and $t_{part}$.
17:                         Calculate SGS stochastic velocity at $x_{part}$ and $t_{part}$.
18:                         Update $x_{part}$ and $t_{part}$
19:                         Save $x_{part}$ to output buffer if $t_{part} >= t_{save}$
20:                     end while
21:                 end for
22:             end for
23:         end while
24:     end for
25:     $T_{L0} = T_{L1}$
26:     Replace first LES data with current LES data.
27: end for
28: MPI: Reduce output data fields from $N$ processes onto process 0.
29: Write model output.
```

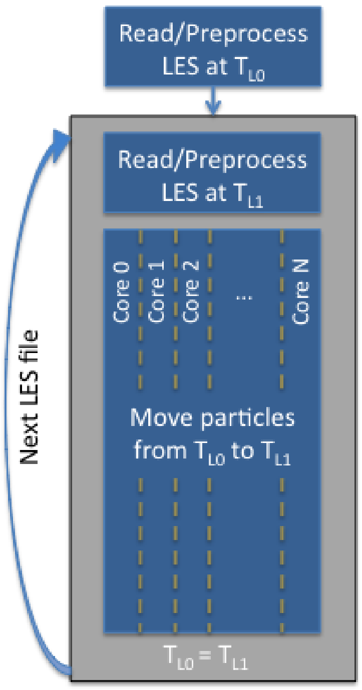

Read/Preprocess LES at T_{L0}
Read/Preprocess LES at T_{L1}
Core 0
Core 1
Core 2
...
Core N
Move particles from T_{L0} to T_{L1}
$T_{L0} = T_{L1}$
Next LES file

Figure 3.5: A detailed description of the computational workflow of the MPI based multi-core LES-LSM, coupled with a simplified diagram which highlights the primary computational factors of the LES-LSM represented as blue boxes (primary factors are “Read/Preprocess LES” and “Move particles from T_{L0} to T_{L1}”). The move particles region of the diagram has been split across the cores of the system using MPI processes. Not shown in the diagram are the additional communication costs incurred by the MPI implementation.

Algorithm 3 OpenMP Multi-core LES-LSM

```
Require: Range of LES files
Require: Particle release schedule (release times, release locations,
    particles per release)
 1: Read and preprocess first LES file valid at $T_{L0}$
 2: for all Other LES files do
 3:   Read and preprocess current LES file valid at time $T_{L1}$
 4:   Calculate the temporal releases with particles between $T_{L0}$ and
      $T_{L1}$.
 5:   Initialize $t_{part}$
 6:   for $Rel_{begin}$ to $Rel_{end}$ do
 7:     while $t_{part} < T_{L1}$ do
 8:       OpenMP: Spawn $N$ threads. Share LES data across
          threads.
 9:       for all Release locations do
10:         for OpenMP: Split particles across $N$ threads do
11:           Get the $t_{part}$ for this particle
12:           while $t_{part} < T_{L1}$ and $t_{part} < t_{save}$ do
13:             Calculate resolved particle velocity at $x_{part}$ and
                $t_{part}$.
14:             Calculate SGS stochastic velocity at $x_{part}$ and $t_{part}$.
15:             Update $x_{part}$ and $t_{part}$
16:             OpenMP: Lock the save statement so no thread
                overlap.
17:             OpenMP: Save $x_{part}$ to output buffer if $t_{part} >=$
                $t_{save}$
18:           end while
19:         end for
20:       end for
21:       OpenMP: Thread join.
22:     end while
23:   end for
24:   $T_{L0} = T_{L1}$
25:   Replace first LES data with current LES data.
26: end for
27: Write model output.
```

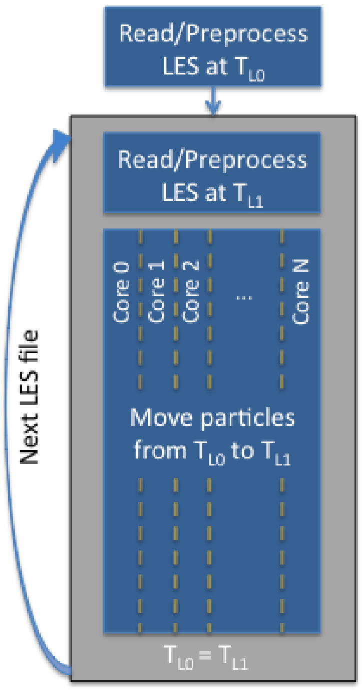

Read/Preprocess LES at T_{L0}
Read/Preprocess LES at T_{L1}
Core 0
Core 1
Core 2
...
Core N
Move particles from T_{L0} to T_{L1}
$T_{L0} = T_{L1}$
Next LES file

Figure 3.6: A detailed description of the computational workflow of the OpenMP based multi-core LES-LSM, coupled with a simplified diagram which highlights the primary computational factors of the LES-LSM represented as blue boxes (primary factors are "Read/Preprocess LES" and "Move particles from T_{L0} to T_{L1}"). The move particles region of the diagram has been split across the cores of the system using OpenMP threads. Not shown in the diagram are the overheads associated with OpenMP data sharing and OpenMP thread spawning.

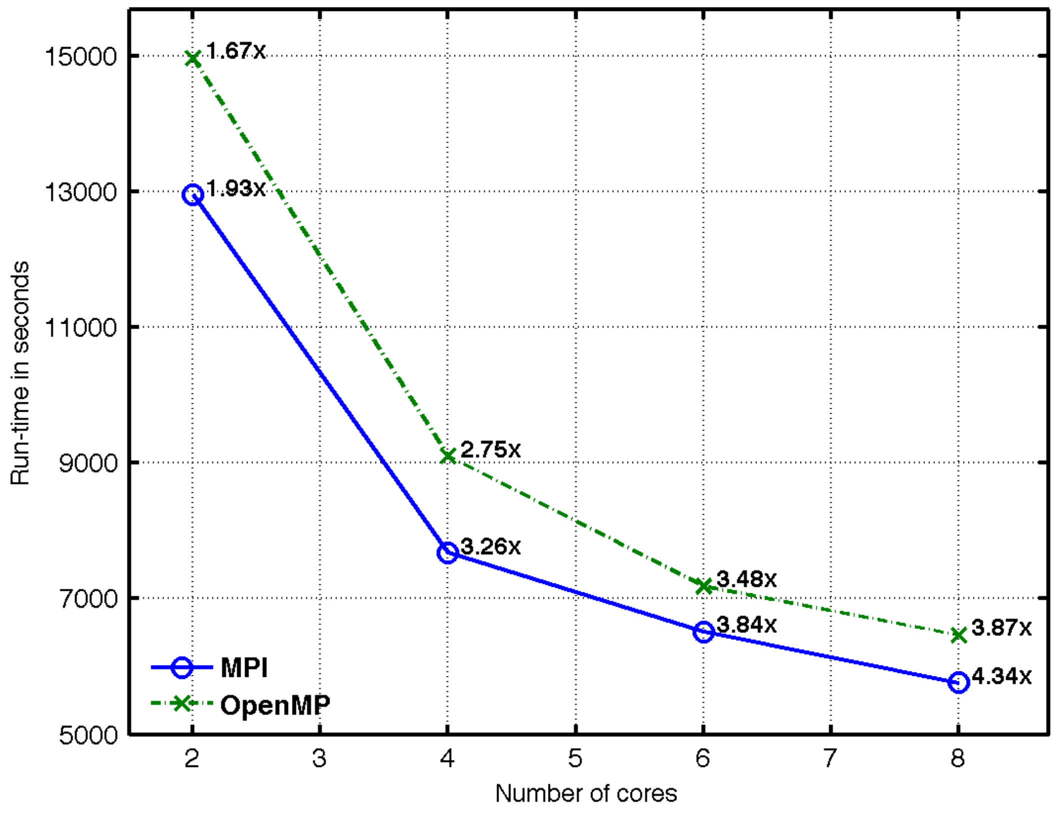


Figure 3.7: The total model runtime, in seconds, of the MPI and OpenMP versions of LES-LSM as CPU cores increases. Speedup factors are embedded in the images with a base sequential performance time of 25,041 seconds. The ave2 workload was used for these runs (described in Section 4.3).

Chapter 4

Methodology

4.1 Hardware & Software

All LES-LSM runs were performed on a desktop work station with 2 quad core Intel Xeon X5450 processors and 8 GB of RAM, running Red Hat 4.1.2-46 and Linux kernel 2.6.18. The X5450 processors have a clock speed of 3.0 GHz, a bus speed of 1333 MHz and a total of 12MB of cache across the four cores. The peak theoretical FP performance of the X5450 was calculated as 48 GFLOP/s [3], by multiplying the per-core peak of 12 GFLOP/s by the number of cores (4). The per-core FP peak performance was calculate by multiplying the clock rate of the core (3.0GHz) by the number of FP instructions that can be issued per cycle(4).

Attached over a PCI-Express x16, 2.0 bus is a single NVIDA GTX285 GPU using NVIDIA driver 190.53. The GTX285 contains 1GB of global memory, runs at 1.476 GHz and contains 240 computational cores organized on 30 multiprocessors. The theoretical peak FP performance of the GTX285 is calculated to be 1062 GFLOP/s by multiplying the GPU clock rate (1.476GHz) by the number of cores on the GPU (240) and by the number of FP instructions that can be issued by a core per cycle (3).

FORTRAN code was compiled using GNU's FORTRAN compiler, gfortran, from the GCC 4.4.3 package. C and C++ code was compiled using gcc from GCC 4.1.2, which is compatible with CUDA and serves as CUDA's compiler for non-kernel C/C++ code. CUDA Toolkit release 2.3 V0.2.1221 was used for GPU builds. Different GCC versions were used for FORTRAN and C++ code because the GCC 4.1.2 version of gfortran contains a bug which caused the binary read operations in the FORTRAN LES-LSM to fail at runtime. The GNU compilers were used in this analysis partly because of conveniences, but also because CUDA's native C/C++ compiler is based around the GNU compiler.

The OpenMP version of the FORTRAN LES-LSM was compiled using the "-fopenmp" option in GCC 4.4.3. The MPI version of the FORTRAN LES-LSM uses MPICH2 version 1.2.1 build with GCC 4.4.3 and using the gforker process manager.

4.2 Performance Metrics

The following metrics are used to describe algorithm performance in this thesis:

- **Application speedup** is calculated as the ratio of the total sequential LES-LSM (single precision) execution time to the total parallel LES-LSM execution time (higher is better). Application speedup will be used to compare the various implementations of the LES-LSM across various workloads.

- **Particle movement bandwidth (particles/sec)** is measured as the number of particles moved through one simulated second, per second (higher is better). This metric allows us to compare the particles movement sections of the different implementations.

- **Mean FP operations per byte** is a way of assessing the total amount of parallelism present in a given application, for a given workload. This metric is an indicator of an algorithm's computational intensity (higher is better) which is helpful to know when moving to a parallel architecture where data will need to be transferred to separate compute devices at some temporal cost. Only the floating point operations involved in particle movement are utilized in this metric.

- **Mean GFLOP/s (billion floating point operations per second)** is another way of assessing the computational complexity of an algorithm, while also accessing how well the algorithm is utilizing its current hardware (higher is better). Only the FP operations involved in particle movement are utilized in this metric. Only single precision arithmetic is considered in this analysis.

- **Mean GFLOP/s % of peak** compares the mean GFLOP/s metric against the peak theoretical GFLOP/s value for the given hardware. Only the FP operations involved in particle movement are utilized in this metric.

4.3 Workloads

The LES-LSM workloads for this analysis are all based on a single batch of LES data containing convective meteorological conditions. The LES data represents 270 time-steps of output, with a simulated Δt of 10 seconds between files. Each file contains 3D double precision (converted to single precision when read into memory) gridded wind fields (u,v,w) and a gridded turbulent energy field (e) with dimensions described in Table 4.1. Two other files per time-step contain meta-information about the data and vertical profiles of turbulent energy variables.

variables	x-dim size	y-dim size	z-dim size	Δx	Δy	Δz	file size
u,v,w,e	256	256	151	20m	20m	12m	316MB

Table 4.1: The dimensions of the 3D wind (u,v,w) and turbulent kinetic energy (e) contained in a single LES file.

Figure 3.2 illustrates the two types of release scenarios used in LES-LSM. The average scenario places particle across the surface plain of the LES data, creating significant demand for memory bandwidth as particles execute load instructions all across the grid. The single release scenario has the potential for better cache utilization because particles are localized within the gridded field, at least in the time immediately following the release. When analyzing the performance of LES-LSM version, it would be valuable to know whether they excel or struggle with certain release scenarios.

There is clearly a value in understanding how the various LES-LSM versions cope with increasing particle counts. The number of particles released in LES-LSM is directly related to its runtime and the fidelity of its simulation results. This is also the parameter that is most likely to be adjusted by a user when he or she is utilizing the model.

Table 4.1(a) describes the configurations of six workloads exercising the two release scenarios and three particle count quantiles. These six workloads, referred to as the realistic workloads because they resemble the original configuration of the model, are used to compare the total computational time of the various LES-LSM implementations. The average release scenario has 81 total release locations distributed in a 9x9 grid. The x and y dimensions of the release location grid will start at grid index $x = 2, y = 2$ and increase independently of one another at increments of $\delta x = 31, \delta y = 31$ grid points until x or y are equal to 250. Because there are 81 locations in the average scenario, the number of particles is a factor of 81. Some parameters that do not vary across the workloads are the number of release (20 releases), the time between releases (100s), the total simulated time (2700s), and the max particle age (1000s). The time-step, δt and write time δt_{save} of the original model were used for all runs.

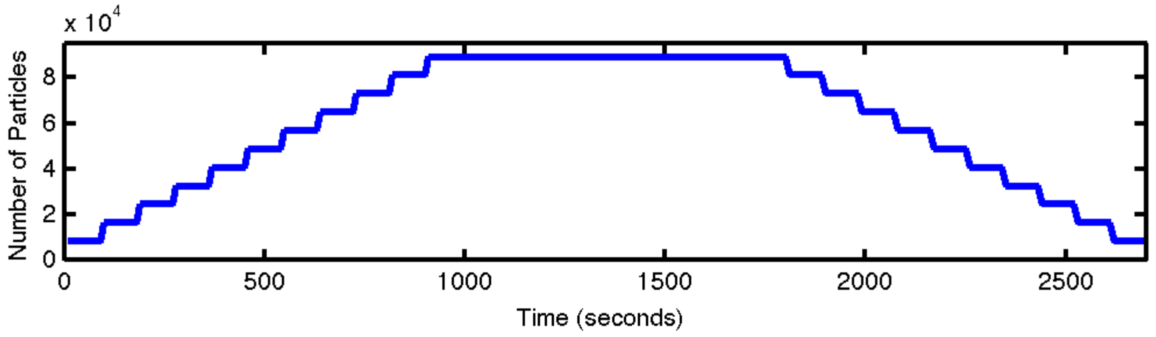
To ease efforts to analyze the inner workings of the different GPU implementations, a set of smaller test workloads were created to exercise the implementations for 900 simulated seconds. These workloads have only two release times and a max particle age of 1000 seconds. Each simulation using a test workload will read 90 LES files, which is enough to clear all the earlier files from RAM so that the next simulation does not artificially benefit from cached LES reads. Table 4.2 describes the test workloads. The test4 workload was used the most in the GPU implementation analyses because it puts sufficient stress on the implementations with 1,620,000 particles in flight through most of the 900 simulated seconds. The smaller tests are used to examine the effects of smaller particle counts in the brute force GPU LES-LSM implementation.

4.4 Sequential Implementation Optimization

In order to assess the true computational benefit of the different LES-LSM variations implemented in this thesis, it is important to establish a baseline sequential implementation that is optimized to the extent possible without algorithm data-flow restructuring. A first pass of the

Workload ID	Release Scenario	Particles Per Release	# Rel.	Release Spacing	Total Parts.	Max Parts. In-flight
ave1	Average	8100 (100 x81)	20	100s	162000	89100
ave2	Average	40500 (500 x81)	20	100s	810000	445500
ave3	Average	81000 (1000 x81)	20	100s	1620000	891000
sin1	Single	8100	20	100s	162000	89100
sin2	Single	40500	20	100s	810000	445500
sin3	Single	81000	20	100s	1620000	891000

(a) LES-LSM Realistic Workloads.



(b) Timeline showing the number of particles per simulated time for workloads ave1 and sin1.

Figure 4.1: Table 4.1(a) describes the 6 realistic workloads used to compare the total computational time of the various configurations of LES-LSM. Figure 4.1(b) illustrates how the 20 time-seperated particle releases with the ave1 workload and the maximum particle age of 1000 seconds affect the total number of in-flight particles in the model at a given simulated time.

Workload ID	Release Scenario	Particles Per Release	# Rel.	Release Spacing	Total Parts.	Max Parts. In-flight
test1a	Average	8100 (100 x81)	2	100s	16200	16200
test2a	Average	40500 (500 x81)	2	100s	81000	81000
test3a	Average	81000 (1000 x81)	2	100s	162000	162000
test4a	Average	810000 (10000 x81)	2	100s	1620000	1620000
test1s	Single	8100	2	100s	16200	16200
test2s	Single	40500	2	100s	81000	81000
test3s	Single	81000	2	100s	162000	162000
test4s	Single	810000	2	100s	1620000	1620000

Table 4.2: LES-LSM Test Workloads

LES-LSM algorithm was done in search of any obvious implementation flaws involving out of order memory access being the main target. One array was restructured to be indexed in cache-optimal order. Also, the memory load operations used to extract the per-particle wind and turbulent energy were re-ordered slightly to make full use of cache locality. With these improvements, a surprising 60% performance speedup was achieved over the original LES-LSM version. A further analysis of the cache-friendly LES-LSM through the GNU profiling tool, gprof, showed that the operations costing the most time were the cached memory load operations. Seeing little room for further improvement, a brief analysis of the compiler optimizations was done.

Aside from the standard optimization levels (-O0, -O1, -O2, -O3) provided by gcc, a few optional optimizations were selected that have the potential for increasing performance in some applications. Table 4.3 shows the performance times of the sequential LES-LSM (single precision) after applying the compiler optimizations one at a time across all the files in the application. A file-by-file compiler flag analysis was not undertaken due to time, though it is likely that further performance could be garnered by doing so. The optional optimizations provide minimal statistically significant benefit over the -O3 optimizations. From these results, it was decided that the "-O3" optimization level would be used for all non-kernel FORTRAN and C/C++ code.

In acknowledging that the GNU compiler is generally not the best compiler for achieving computational performance, a pair of sequential LES-LSM (single-precision) runs using the ave2

Optimization	**Run-time (seconds)**
-O0	1925
-O1	1224
-O2	1205
-O3	1146
opt_O3_merge_all_constants	1142
opt_O3_profile	1137
opt_O3_unroll3	1141
opt_O3_unroll6	1139

Table 4.3: Exploration of the optimization space of the sequential LES-LSM version using an abridged workload. Optimizations applied to the entire application.

workload were performed on a different machine (one which had Portland Group licenses installed) using GNU compilers (gfortran) and Portland Group compilers (pgf90). Using the same "-O3" optimization flag in both versions, the Portland Group version performed 10% faster than the GNU version. As this was such a simplistic analysis, it is unclear which parts of the LES-LSM are preforming better in the Portland Group version than the GNU. What is clear, however, is that the LES-LSM performance results derived from the GNU compiler are far from optimal, and that fact should be taken into consideration when comparing the sequential LES-LSM with the GPU versions of the LES-LSM.

4.5 GPU Implementation Optimization

Optimizing the GPU versions of LES-LSM was an exercise in following the CUDA Best Practices guide [12] and making the best use of the different memory regions on the GPU. To compare one optimization avenue with another, timing statements were added using OpenMP timing commands with micro-second accuracy. In the case of asynchronous kernel invocations and memory read operations, the CUDA profiler was used occasionally to determine how long such operations were actually taking.

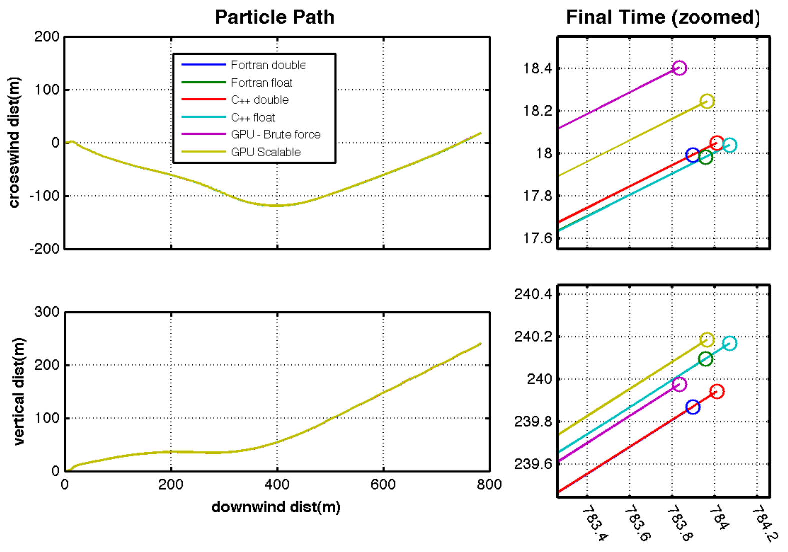
4.6 Correctness Across Versions

Confirming correctness is a difficult task for an application as large and as research centric as LES-LSM. The fact that different random number generators are used across the various implementations means that dispersion results can never be identical. As a substitute for comparing the final LES-LSM output fields for correctness, each version of the LES-LSM was instrumented to use the same set of random numbers generated by the sequential LES-LSM (double precision). Figure 4.2 shows the results of carrying 3 particles, released at different locations, through 250 simulated seconds using the various version of the LES-LSM. The final particle locations are within a margin that would be expected after the approximately 3.6 million FP operations needed to move a particle through 25 LES files. The concentration field results in Figure 4.3 show that the sequential LES-LSM and the GPU variants of the LES-LSM are in close agreement with one another when different random number generators are used in each version.

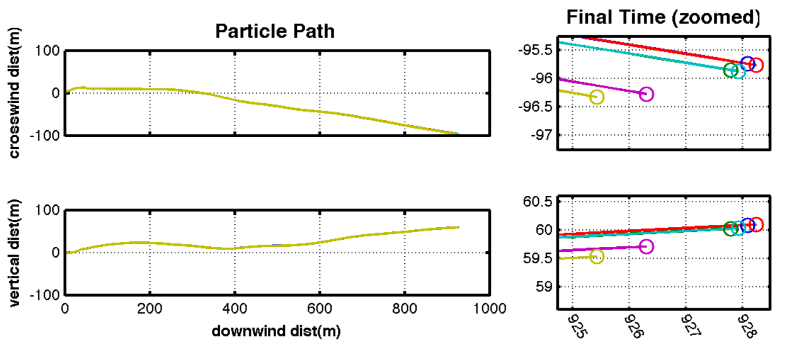
4.7 Implementation Performance Comparison

Having optimized the various versions of LES-LSM and confirmed some measure of correctness between versions, computation time can be analyzed. In the case of LES-LSM, the most important metric of performance is total runtime of the model. It would also be useful to have some specific information about where time is spent in the application. For this reason, each version of LES-LSM was instrumented with timing calls that can be activated or deactivated at compile time using compiler preprocessor directives. Timers write easily identifiable statements to standard output which can be parsed by an analysis script. Table 4.4(a) describes the specific operations that will be timed and any supplementary data associated with that operation.

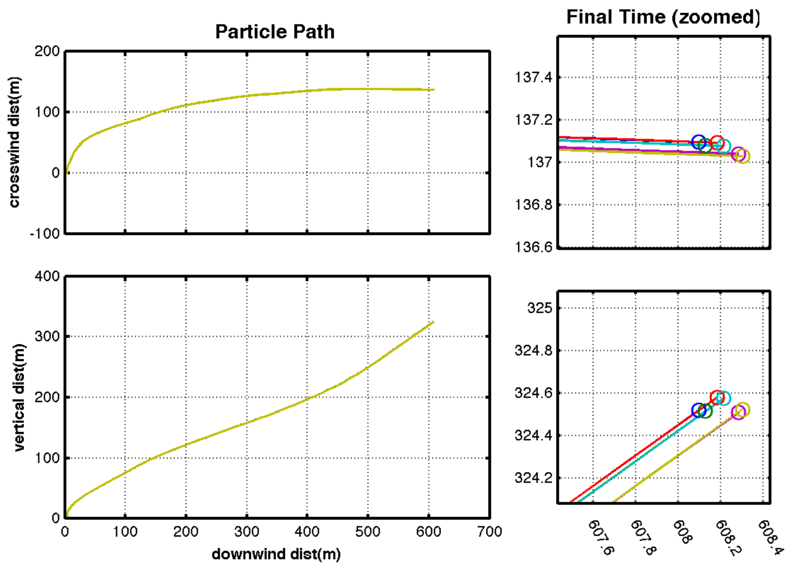
A set of scripts were written to allow multiple LES-LSM runs to trigger in succession while collecting runtime information and standard output streams into an organized directory structure. The user can specify whether to use the version with timing instrumentation or the the timer free version. The user may also specify the order to run versions, and number of times a particular



(a) Particle One

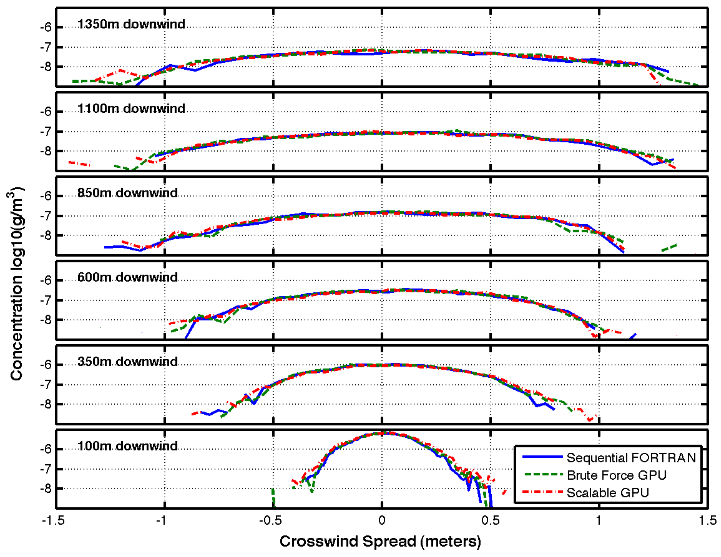


(b) Particle Two

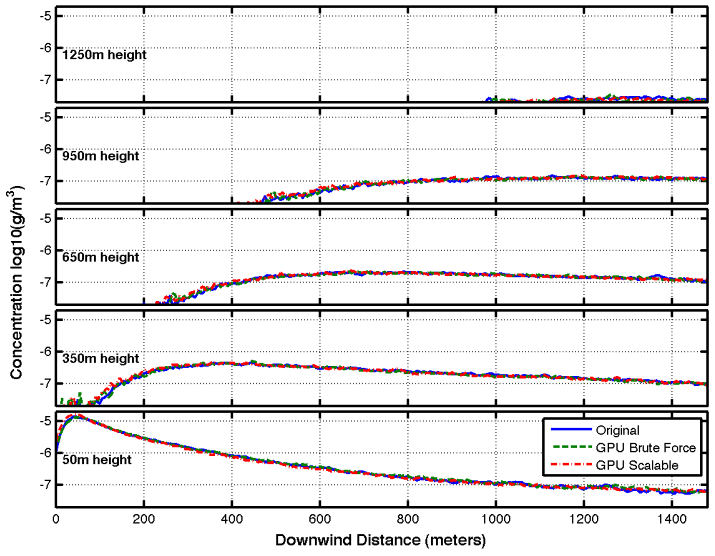


(c) Particle Three

Figure 4.2: The paths of three particles, simulated for 250 simulated seconds across the different LES-LSM implementations using the same random number values in each version. All variation in final particle location are due to differences in FP behavior.



(a) LES-LSM Correctness Across Implementations Crosswind



(b) LES-LSM Correctness Across Implementations Downwind

Figure 4.3: Release concentration results of the original LES-LSM and the two GPU LES-LSM implementations using the ave1 workload. Figure 4.3(a) illustrates the crosswind concentration at various downwind distances at a height of 50 meters. Figure 4.3(b) shows the maximum downwind concentration at various heights.

Action To Time	**Supplementary Information**
Application startup time.	
Reading file from disk.	Bytes read.
Preprocessing read contents.	
Time moving particles.	Number of particles.
Application closing time.	

(a) General Actions to Time

LES-LSM Version	**Timer Utility**
FORTRAN Sequential	MPI_Wtime()
GPU Versions	omp_get_wtime()

(b) Timer Commands Per Version

Figure 4.4: General Timing Information

LES-LSM should be run. Total runtime is recorded by calling the tcsh shell command "date +%s" before and after invoking the model to print the current time in seconds since since the linux epoch time. A set of MATLAB scripts were written to traverse the directory structure generated by the batch scripts for relevant information.

4.8 GPU Implementation Analysis

Aside from the timing information described in Table 4.4(a), GPU specific information can be helpful in comparing the GPU version against a CPU version. CUDA's profiler provides timing information about memory transfers and kernel runtimes. The GPU programs were also instrumented to record the information shown in Table 4.4.

Action To Time	**Supplementary Information**
Host to device memcpy.	Bytes transfered. CUDA Stream number.
Move particles using device.	Number of particles. CUDA Stream number.
Extra GPU processing steps (if any)	

Table 4.4: GPU Actions to Time

The timed versions of the GPU LES-LSMs were run against the test workloads to derive simple models of expected computational performance across various workloads. The exercise of creating a performance model is valuable not only because it forces one to calculate all the statistics relevant to moving particles through the LES data, but because it provides a way of verifying that the statistics closely correlated with actual performance times. The performance models make it possible to examine the theoretical bounds of large and small problem sizes that were beyond the scope of the workloads used in this thesis. Performance models can also be useful when trying to adapt a model to fit within operational time and workload constraints.

Chapter 5

Brute Force CUDA Implementation

The initial effort to utilize CUDA parallelism into LES-LSM is referred to as "brute force" because it is a direct translation to C/C++ from the original FORTRAN model, preserving its control and data flow paths. This effort was meant to represent an approach with minimal investment in developer time, and a focus on scientific results over engineering efficiency. However, the decision to use C and C++ for the brute force implementation is certainly not a minimal development task. In a truly minimal development effort, the most direct way to use CUDA would be by calling external C function from the original FORTRAN model. The hope is that the performance of the C/C++ brute force implementation will still have a computational workload resembling that of a FORTRAN equivalent using external C calls for CUDA optimization.

5.1 Sequential C/C++ Implementation

The first step in bringing LES-LSM closer to CUDA capabilities was to manually convert the sequential application to C and C++. The approach of coupling the original FORTRAN based LES-LSM with external C based CUDA functions was not considered because it would have been a major encumbrance on upcoming efforts to restructure the application to better suit GPU computing parallelism. The intended future uses of the LES-LSM provide additional justification for the conversion from FORTRAN to C/C++. Recent advanced FORTRAN programming tools like the Portland Group's CUDA FORTRAN were not in production at the outset of this thesis, and were thus not considered [19].

The C/C++ version was derived directly from the original code with several changes to account for the language differences. C++ classes were used to encapsulate the file reading operations, multi-dimensional array management and various other data flow management capabilities. The decision to use C++ classes, as opposed to C functions, was made purely out of a desire to follow proven engineering practices related to object-oriented analysis and design. In moving from FORTRAN, byte swapping C functions needed to convert big-endian LES output numbers to native little-endian byte ordering. The random number generator used in the original LES-LSM was replaced with a simple linear congruential random number generator. Parameter values, which were "hard coded" inside the original LES-LSM, were moved into parameter files.

Table 5.1 shows the computational performance results of the C/C++ sequential LES-LSM implementation using single precision FP operations. Comparing these results with the sequential FORTRAN LES-LSM (single precision) in Figure 3.1, there are clear differences. The LES preprocessing step takes an order of magnitude longer in the C/C++ version, than the FORTRAN version. Also, particle movement in the C/C++ version is almost twice as slow as the FORTRAN version. The reasons for these discrepancies was not researched, beyond ensuring that it was not related to cache misuse. We theorize that they are a result of a combination of pointer dereferencing, C++ virtual method table lookups and poor compiler optimization in the more complicated C/C++ code.

		Workloads					
		ave1	ave2	ave3	sin1	sin2	sin3
Workload description	Total particles	162000	810000	1620000	162000	810000	1620000
Total time per operation	LES read	954s	961s	961s	962s	961s	961s
	LES preprocess	389s	390s	393s	389s	389s	388s
	Move particles	9215s	43968s	87511s	8381s	42000s	83717s
	Miscellaneous	4s	5s	6s	4s	4s	5s
	Total time	10563s	45325s	88871s	9737s	43355s	85072s
Percent of total time	LES read	9.03%	2.12%	1.08%	9.88%	2.22%	1.13%
	LES preprocess	3.68%	0.86%	0.44%	4.00%	0.90%	0.46%
	Move particles	87.24%	97.01%	98.47%	86.08%	96.87%	98.41%
	Miscellaneous	0.04%	0.01%	0.01%	0.04%	0.01%	0.01%
FP statistics	Mean FP ops per byte	< 55.0	< 274.9	< 549.7	< 55.0	< 274.9	< 549.7
	Mean GFLOPS	0.243	0.249	0.247	0.259	0.259	0.259
	Mean GFLOPS % of peak	2.023%	2.079%	2.057%	2.160%	2.159%	2.159%

Table 5.1: Single precision FP computational performance results from the sequential C/C++ LES-LSM. GFLOPS statistics were calculated using the FP complexity observations from Section 3.3. The Mean GFLOPs % of peak metric uses the single core CPU peak of 12 GFLOPs.

5.2 Brute Force GPU Implementation

The CUDA brute force model was implemented by replacing the two inner-most **forall** loops with a single call to a CUDA kernel to move the particles as show in Algorithm 4. In addition to the kernel invocation, many data preparation and data post-processing steps were needed to make the CUDA implementation. Generally they can be grouped into 3 categories. 1) Prepare the device and device memory to run the LES-LSM kernel. 2) For each LES file, copy the LES data to the device and bind textures references to the LES data regions. 3) Copy down the results to the host and release the device.

The CUDA memory usage and kernel implementation details of the brute force GPU LES-LSM are described in the following sections.

Global Memory

All arrays relating to the state of the particles are stored in global memory. Arrays are ordered such that the number of particles per release location is the fastest varying dimension. In this way, the 3D particle location array for a given release location and release time would be organized in memory as $[x_{p1}, ..., x_{pN}, y_{p1}, ..., y_{pN}, z_{p1}, ..., z_{pN}]$. This ordering will become valuable when kernels access the arrays on a per-particle basis. The LES vertical profiles are also stored in global memory with no change to ordering from the CPU arrays.

Texture Memory

The LES data variables (u,v,w,e) for times T_{L0} and T_{L1} are saved as 3D textures. The cache associated with texture memory greatly helps the total application performance for the LES-LSM kernel because though particles do not access the LES volumes in an ordered fashion, the particles do share some spacial locality within the LES volume which the texture cache utilizes.

Algorithm 4 Brute Force GPU LES-LSM

```
Require: Range of LES files
Require: Particle release schedule (release times, release locations,
    particles per release)
 1: CUDA: Select the device.
 2: CUDA: Define memory regions on device.
 3: CUDA: Copy constants to device constants memory.
 4: Read and preprocess first LES file valid at $T_{L0}$
 5: CUDA: Synchronous copy $T_{L0}$ LES to device texture memory.
 6: Initialize $t_{part}$
 7: for all Other LES files do
 8:   Read and preprocess current LES file valid at time $T_{L1}$
 9:   CUDA: Synchronous copy $T_{L1}$ LES to device texture memory.
10:   Calculate the temporal releases with particles between $T_{L0}$ and
      $T_{L1}$.
11:   Get the $t_{part}$ for this particle
12:   for $Rel_{begin}$ to $Rel_{end}$ do
13:     while $t_{part} < T_{L1}$ do
14:       CUDA: Create block size to contain all particles released
          at $Rel_{current}$.
15:       CUDA: Move particles until $t_{part} >= T_{L1}$ or $t_{part} >= t_{save}$
16:     end while
17:   end for
18:   $T_{L0} = T_{L1}$
19:   Replace first LES data with current LES data.
20:   CUDA: Replace first LES data with current LES data on device.
21: end for
22: CUDA: Copy output grids from device.
23: Write model output.
24: CUDA: Free device memory and release the device.
```

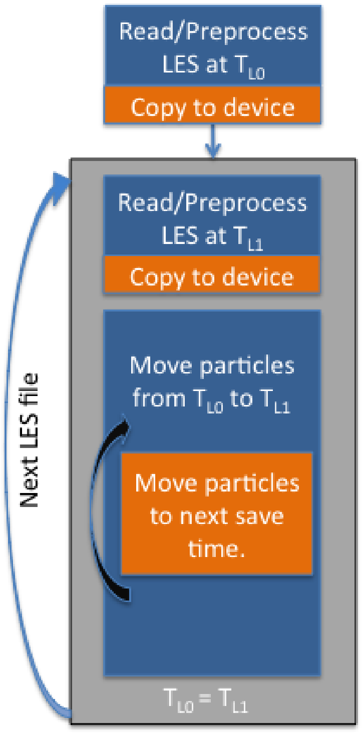

Read/Preprocess LES at T_{L0}
Copy to device
Read/Preprocess LES at T_{L1}
Copy to device
Move particles from T_{L0} to T_{L1}
Move particles to next save time.
Next LES file
$T_{L0} = T_{L1}$

Figure 5.1: Detailed textual workflow the brute force GPU LES-LSM along with condensed flow diagram. The flow diagram highlights the most time consuming operations including reading and preprocessing LES data, copying LES data to the GPU device and moving particles.

Constant Memory

Variables used across all kernels and pointers to the global memory arrays are stored in constant memory as C structs.

Local Memory

CUDA unfortunately has trouble with structs and statically defined arrays inside of kernels. The CUDA compiler generally places these types of variables into local memory. As a consequence, many of the internals of the LES-LSM kernel are placed into local memory.

Shared Memory

Some of the most frequently accessed static arrays, like the arrays used to hold the particle location and velocity are loaded into shared memory at the start of the kernel to avoid the overhead associated with local memory. The vertical profiles for the LES times are also loaded into shared memory to minimize the cost of repeatedly accessing global memory in a non-uniform pattern.

Register Memory

The CUDA compiler is in charge of assigning values to registers and reporting to the user how many registers were used. Unfortunately, the LES-LSM particle movement kernel requires 100+ registers for each thread. This means we are effectively limited to running 128 threads per block, out of the maximum, 512. From this information, the CUDA occupancy calculator estimates that 13% of the warp is occupied. The CUDA compiler has options which allow a user to suppress the number of registers assigned to each thread, but it does so by spilling register values to local memory. For the LES-LSM kernel, little benefit was found in suppressing register counts to increase the number of threads per block.

Kernel

The LES-LSM kernel is invoked using blocks that only vary across the x dimension, which corresponds the number of particles. The kernel begins by loading the particle location and wind information from global memory into shared memory buffers. Because of the per-particle organization of these arrays in global memory, all of the reads for each warp are coalesced. Also, each thread helps to load the LES vertical profiles from global memory to shared memory, in an order which ensures that the reads are coalesced. Each particle now runs through the processing steps of computing the resolved and stochastic velocity components and stepping the particle location forward. When a particle reaches a writing condition, it increments the output grids using CUDA atomic adds to avoid problems with writing to the same location from multiple threads. Finally, when an ending condition is reached, the shared memory particle location buffers are written back to the global memory regions.

5.3 Analysis

A summary of the brute force GPU LES-LSM's performance across the realistic workloads is shown in Table 5.2. Logically, the LES read and preprocess operations take approximately the same amount of time as in the sequential C/C++ LES-LSM. The particle movement section, however, has been significantly improved relative to the sequential C/C++ LES-LSM by the use of our GPU. The use of the GPU also incurs some data transfer costs, highlighted in the copy to device operation, and other costs relating to allocating/deallocating GPU memory and copying values from the GPU, all categorized under the miscellaneous operation.

The performance of the brute force CUDA implementation was further analyzed using the results from a single run, instrumented with timing print statements, on each test workload described in Table 4.2. The results were processed with a focus on defining a model of the computational performance of the brute force CUDA implementation. The performance model has the same structure as the algorithm described in Algorithm 4, but instead of moving particles, it simply increments a

		Workloads					
		ave1	ave2	ave3	sin1	sin2	sin3
Workload description	Total particles	162000	810000	1620000	162000	810000	1620000
Total time per operation	LES read	970s	978s	977s	970s	977s	979s
	LES preprocess	392s	391s	391s	390s	389s	389s
	Copy to device	19s	19s	19s	19s	19s	19s
	Move particles	283s	1070s	2042s	249s	995s	1923s
	Miscellaneous	3.924s	3.991s	3.878s	3.872s	3.672s	2.571s
	Total time	1671s	2466s	3438s	1636s	2388s	3319s
Percent of total time	LES read	58.02%	39.67%	28.43%	59.29%	40.92%	29.50%
	LES preprocess	23.45%	15.84%	11.39%	23.85%	16.29%	11.72%
	Copy to device	1.16%	0.78%	0.56%	1.18%	0.81%	0.58%
	Move particles	16.94%	43.40%	59.39%	15.21%	41.66%	57.94%
	Miscellaneous	0.43%	0.30%	0.23%	0.47%	0.33%	0.25%
FP statistics	Mean FP ops per byte	< 55.0	< 274.9	< 549.7	< 55.0	< 274.9	< 549.7
	Mean GFLOPS	8.208	8.208	8.208	8.208	8.208	8.208
	Mean GFLOPS % of peak	0.773%	0.773%	0.773%	0.773%	0.773%	0.773%

Table 5.2: Single precision FP computational performance results from the brute force GPU LES-LSM. GFLOPS statistics were calculated using the FP complexity observations from Section 3.3. The Mean GFLOPs % of peak metric uses a peak of 1062 GFLOPs for the GTX285 GPU.

time counter to estimate the total runtime.

Table 5.2(a) shows the timing values of the most time consuming operations in the brute force CUDA implementation. Many of the operations do not vary across the test workloads or any of the workloads used in this thesis because they are solely dependent on the size of the LES volume. These operations all have unimodal distributions and are modeled as mean values. The movement kernel time must be modeled as a function of the number of particles. An understanding of the particle movement kernel tells us that its computation time should be linearly related to the number of particles being moved. Thus, least squares linear regression was used to calculate a linear model for movement kernel time, with a few caveats. One caveat is that the collection of particle movement times show a slightly bimodal distribution. The two modes represent the average and single release scenarios which perform differently because the single release scenario benefits from better warp level locality on the GPU. Another caveat is that the particle movement time distributions contains a tail in the positive time direction which is explained by the LES-LSM's use of a shorter δt time-step when a particle is in an area of very high turbulence.

The performance model validation results shown in Figure 6.8(b) tend to under-predict the computation time for average release scenarios and over-predict the computation time for single release scenarios. The discrepancy is an artifact of grouping the average and single release kernels together. As in the sequential model, the single release scenario benefits from greater particle

Operation	Time Mean (sec)	Time Variance (sec^2)	# Samples
Per run: startup	0.8477	1.5836$*10^{-4}$	8
Per run: closing	0.8254	2.386$*10^{-4}$	8
Per LES: read	3.5371	1.1891$*10^{-2}$	712 (89 per workload)
Per LES: preprocess	1.4613	4.6008$*10^{-4}$	712 (89 per workload)
Per LES: copy to device	0.0728	1.1774$*10^{-6}$	712 (89 per workload)
	Linear Model		Bandwidth
movement kernel	$1.1764 * 10^{-6}(sec/part) * N_{particles} + 0.0040(sec)$		850,020 (part/sec)

(a) Brute Force CUDA Performance Model Parameters

Workload:	ave1	ave2	ave3	sin1	sin2	sin3
Observed (sec):	1666	2448	3417	1634	2371	3298
Predicted (sec):	1647	2426	3399	1647	2426	3399

(b) Brute Force GPU LES-LSM Performance Model Error

Figure 5.2: Performance model and validation results for the brute force GPU LES-LSM. Table 5.2(a) contains invariant model parameters, which occur once every run or LES file, represented as a mean and variance. The particle movement operation is represented as a linear model of the number of particles to be moved for one simulated second. The model parameters are validated against the realistic workloads in Table 6.8(b).

locality and better cache utilization (texture cache) than the average release scenarios.

Chapter 6

Scalable CUDA Implementation

The brute force implementation works well at accelerating the LES-LSM for the workloads analyzed in this thesis. However, there are limitations to the brute force implementation which could require some algorithm re-design for future workloads. Specifically, the limited amount of memory on a GPU could cause problems if LES data-sets become much larger. The two LES data increments used in the brute force implementation, in single precision format, occupy 317MB of our 1GB GPU. If the LES volume were to increase by 50% in each dimension, it would no longer fit in the 1GB card. The available memory space on GPU hardware tends to increase with each new generation of chips, so perhaps this is not a major problem. But it does present a scenario which could cause operational problems at a later date.

Another limitation to the current LES-LSM model is that it does not keep the problem on the GPU for as long as possible. The brute force implementation only carries the particles for 3 δt time-steps until a writing point or the end of the LES volume is reached. A better implementation would keep more of the problem on the GPU. Perhaps an ideal application would even have multiple LES volumes on the GPU at once so that a kernel could carry particles even farther in one kernel invocation.

A third LES-LSM limitation is that the entire LES volume needs to be read into memory even if a problem only uses a small portion of the volume. Most of the workloads used in this analysis make use of the majority of the LES volumes, except at the start of a simulation when all the particles are close to the surface. However, a later version of the LES-LSM might prefer to save computational effort by reading only the subset of meteorological data needed for computation.

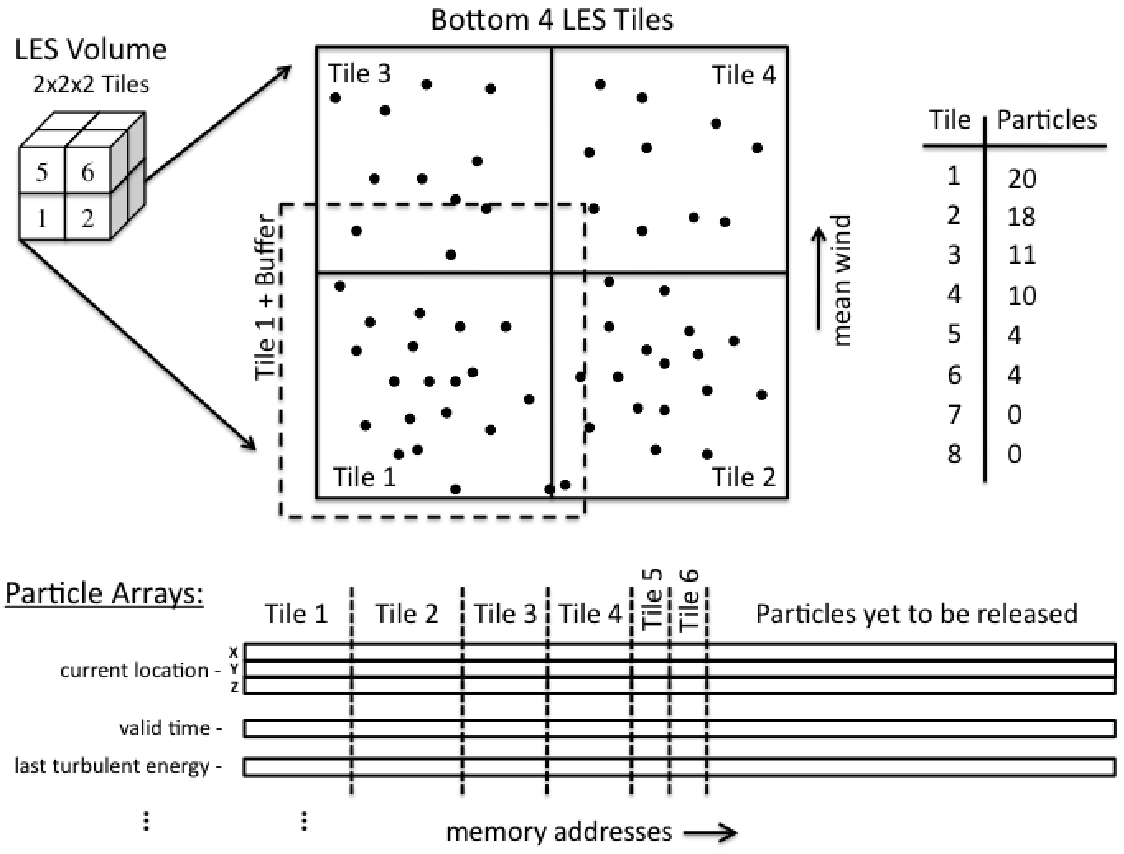
Finally, an advanced LES-LSM version might benefit computationally from being able to distribute work across multiple GPUs. This would be particularly useful if the number of particles is very large and the total model run-time is bound by GPU computation time. The following sections outline the structure of an LES-LSM model that is scalable to larger LES domains, overlaps LES data reading with GPU computation, and could be easily extended to use multiple GPUs.

6.1 Implementation

The approach taken in enhancing the LES-LSM model for scalability involves breaking down the problem space into geographically smaller work units which require less overall memory, and can fit in the memory space of the GPU of choice. The simplest way to decompose the problem geographically is to tile the LES data volume along the X, Y, and Z axes to create a series of 3D, sub-boxes. A more sophisticated technique might attempt to dynamically seek out the location of particles and attempt to minimize the total number of 3D tiles. Our implementation uses the tiling decomposition strategy, but with the software flexibility to add new problem decomposition algorithms easily.

Assigning particles to tiles is done entirely on the device by first running a kernel to assign tile index values to each particle based on its global location. Next, a fast (O(kn) efficiency) radix sort from the CUDA SDK is used to sort the particles by tile number in GPU memory. Finally, a reorganization kernel is called to reorganize all the device arrays associated with individual particles to match the tile number ordering. The host can then request the array of tile numbers from the device and create a series of work units consisting of the geographic 3D tile and the range of particles contained in the tile. Figure 6.1 shows an example of particles being reorganized in memory by their tile number. Because this is only a single GPU implementation, all of the particle information is contained on the single device and no inter-device communication is needed.

To cope with particles traveling outside of their 3D tile over the course of particle movement, a certain amount of buffer zone, also referred to as ghost zone, is needed along each dimension.



Tile	Particles
1	20
2	18
3	11
4	10
5	4
6	4
7	0
8	0

Figure 6.1: The top portion of this figure depicts a 3D LES volume which has been sliced into 8 tiles by splitting each dimension in half. An overhead view of the bottom 4 LES tiles shows an assortment of particles in each tile, with a particle count for each of the 8 tiles on the right. The box with dashed lines around tile 1 represents the manually configured buffering region around each tile which has been configured to account for a south to north mean wind. The particle arrays at the bottom of the figure show how the particles of each tile are sorted together on the device in preparation for the particle movement kernel.

This zone size is left for the user to specify through parameter files at the start of the run. A good choice of buffers will take into consideration the mean wind vector of the time-series of LES data to minimize excess memory handling. Figure 6.1 illustrates a buffer zone which grants extra buffer area in the mean downwind direction. If a particle still travels beyond the extents of the buffered 3D tile, the CUDA thread responsible for that particle halts before completion time, storing the latest location and valid time of the particle. Particle halts, henceforth called off-grid misses, are computationally costly because they require LES volumes to be repeated. Buffer zones that surpass the X,Y boundaries of the LES volume are populated by rolling over to the opposite sides of the X,Y dimensions. Buffer zones in the Z dimension that exceed an LES volume boundary are added to the buffer on the opposite side. With these rollovers comes a restriction that a single point in the LES volume cannot correspond to multiple points in the 3D tile. An example of this restriction being violated is if our 3D sub-grid was simply the full LES grid, and a buffer of 10 pixels was requested in each dimension.

With the problem decomposed into smaller work units, the simulation is freed from the restriction of moving particles only from T_{L0} to T_{L1}. Opening the problem to larger spans of simulation time means that kernels can spend more time processing work without interruption, thus reducing some of the overhead of kernel invocations. However, this comes at the cost of needing larger buffer regions around the 3D tile to prevent off-grid misses, and at the cost of having to perform more LES file reads before invoking a kernel.

CUDA offers three features which help to offset the added costs of reading multiple LES data times and requiring larger buffer zones. The first is the fact that CUDA kernels run asynchronously relative to the CPU, meaning CPU threads can be occupied performing read operations to prepare work units while the GPU runs a kernel on an earlier work unit. However, the GPU will still be left idle while the CPU reads the first work unit. The second feature to help offset reading overheads, CUDA streams, allow data transfer to or from the device while a kernel is running on that same device. With this feature, the host process can be staging the upcoming work units to the GPU while the GPU is processing an earlier work units, thus hiding the transfer cost. Finally, CUDA's pinned memory capabilities provide greater host to device transfer bandwidth, but must be used in limited supply so as not to cause premature paging in the host memory system.

Figure 6.2 illustrates the roles of the two host threads of the LES-LSM in moving data to the device for computation. Assuming that a list of work units has already been created, host thread 2 will be spawned with the sole purpose of requesting LES meteorology data from the data server and populating the work queue with work units ready for processing. The original host thread, labeled host thread 1, then removes items from the work queue when a device stream becomes available. When a stream is available and a work unit ready, the work unit meteorology is copied to the device asynchronously and the particle movement kernels invoked.

To represent all of these configurations, some variables are defined. The size of the desired particle movement time-span will be specified by N_{times} as the number of LES files to span. A span from time T_{L0} to T_{L4}, representing 40 seconds worth of meteorology, is representative of an N_{times} of 5. The number of buffers in the Work Request Manager is specified by $N_{workbuffers}$. The number of streams used to transfer data to the host and run kernels is specified by $N_{streams}$. Finally, the LES tiling parameters will be described as a 3 digit number represented by the variable P_{tile}. Each digit of the tiling parameters represents the number of tiles per dimension in order of X,Y and Z. A P_{tile} of 334 has three tiles along the X and Y axes, and four tiles along the Z axis for a total of 36 tiles.

The following sections describe the inner workings of the LES data server, the work request manager, and the particle movement kernel.

6.1.1 LES Data Server

The LES data server is responsible for fulfilling requests for 4D tiles of data from the collection of LES data on disk, where the fourth dimension of the requests is the number of LES time increments. Depending on how the user has configured the model, a practical number of times is in the range of 2 to 8 LES times per request. The simplest approach to meeting such requests would be to read the full LES grids for all the requested time, then copy the relevant 3D spacial

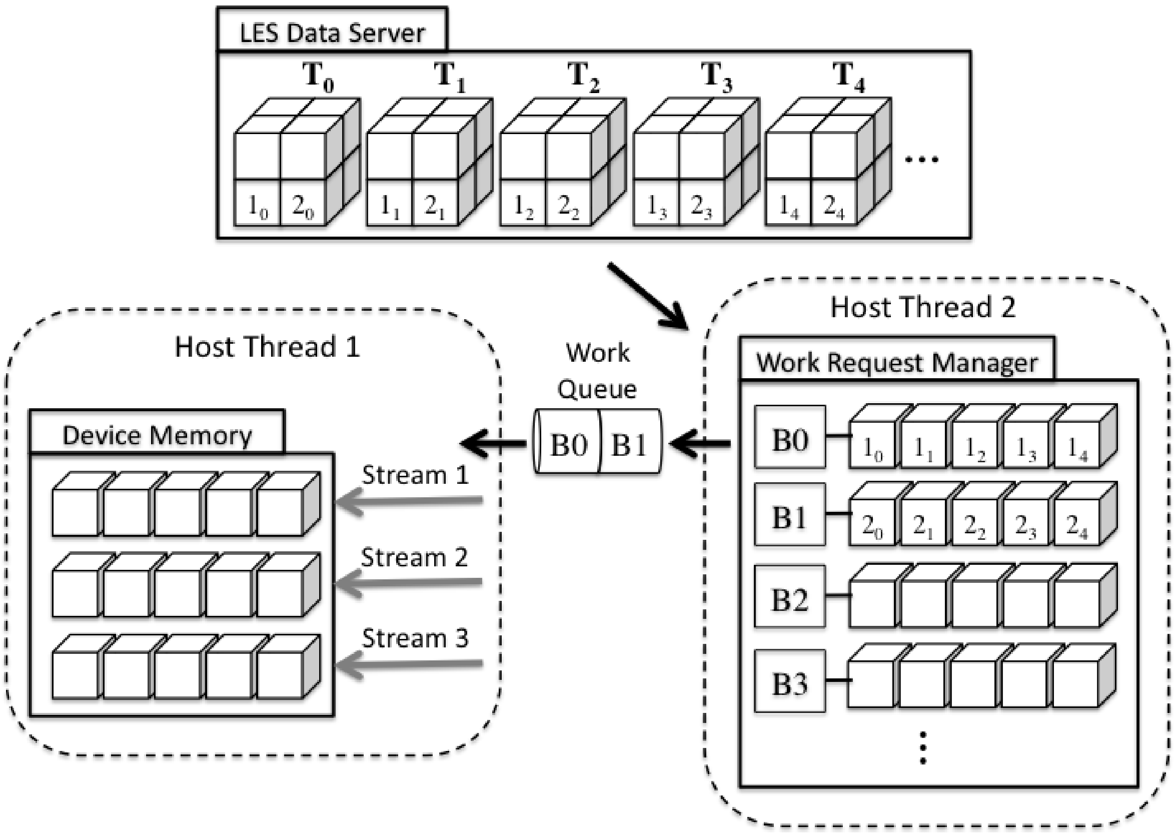


Figure 6.2: The Work Request Manager on host thread 2, responsible for loading data from the LES Data Server, has already populated buffers B0 and B1 with data from LES tiles number 1 and 2 and time values T_0 to T_4. References to buffers B0 and B1 have been added to the Work Queue where host thread 1 is responsible for feeding the buffer contents to the GPU device over one of the three CUDA streams defined on this device.

sections of the data to the requester. A more advanced approach might be to read only the relevant 3D sections from each requested time file. Our approach was the same as the latter, but with the inclusion of a buffering scheme, that will buffer the full LES grid contents for all of the requested times.

The buffering scheme works by first creating a pool of N_{times} buffers with each buffer sized to contain the full contents of an LES file, including all 4 meteorological products (u,v,w,e). When a request arrives, unused buffers will be retrieved from the pool and assigned an LES time to represent. As the data server reads 3D tiles from the LES files, the 3D tiles are placed into the buffers at their proper relative locations in the LES volume. When all the disk reads are complete, the request is serviced by copying the 3D box from the data server buffers.

The buffered approach helps in achieving better computational performance by avoiding redundant LES file reads of the spacial area in the LES volume. This is a very common occurrence because of the user specified ghost zones surrounding each 3D work tile. The buffer pool approach helps performance by avoiding repeated allocation and deallocation of memory space.

6.1.2 Work Request Manager

The work request queue is responsible for allocating and populating the work units that will be transfered to the GPU. Given the tile parameters, P_{tile}, and the number of LES times, N_{times}, the work request manager allocates $N_{workbuffers}$ 4D tiles for each variable (u,v,w,e) of CUDA pinned memory at startup. When it comes time for computation, a new host thread is created for the work request manager using OpenMP. On the new thread, a list of work unit requests is received at the work request manager where they are processed as fast as possible in a tight loop. Work units are populated by requests to the LES Data Server. The loop will continue making requests of the LES Data Server until either all requests have been satisfied, or all $N_{workbuffers}$ are occupied.

As work buffers become occupied, a reference to the work buffer entry is placed in a queue for a separate processing thread to retrieve. Locking operations provided by the pthreads library were used to implement thread safety in the work request manager. When the processing thread is finished with a work buffer entry, that buffer is made available for further work units requests. This multi-threaded approach makes it possible to simultaneously read requests and processes requests.

6.1.3 Particle Movement Operations

The particle movement host thread runs simultaneously with the work request manager thread where it waits in a loop for prepared work units to appear in the work queue. Upon arrival of a work unit, that unit is assigned to a CUDA stream where it is asynchronously copied to the device and the particle movement kernel is called asynchronously. From the host perspective, these operations are instantaneous, so the host can continue servicing other work units. The particle movement thread is also responsible for polling the CUDA streams to determine when a work unit has finished and can be released. The scalable GPU LES-LSM makes use of the GPU resources in the following way.

Global Memory

As in the brute force implementation, all arrays relating to particle state are located in global memory with particle index as the fastest varying dimension to make coalesced reads and writes possible. The only other arrays defined in global memory is a single vertical profile of kinetic energy information which is read once per thread, per time-step. All other vertical profiles, which are accessed more often, are stored in shared memory.

Texture Memory

Texture memory is used to store the LES meteorology data of the work unit tile. To get the 4D tile of data into a 3D texture, the third dimension of the texture contains the vertical Z height dimension and the time dimension by stacking times on top of one another.

Constant Memory

Constant memory is again used primarily for numbers that are constant over the LES-LSM simulation, and for references to arrays in global memory.

Local Memory

As with the brute force implementation, some structures and statically defined arrays are offloaded to local memory for lack of a better location.

Shared Memory

Shared memory is used as a buffer for some of the most frequently accessed particle state arrays that are located in global memory. Also, four profile arrays are placed in shared memory to reduce the cost of reading them from global memory.

Kernel

The scalable LES-LSM kernel runs with a block size of 128 threads (achieving 13% warp occupancy according to the CUDA occupancy calculator), where a thread is synonymous with a single particle. Normal sized kernels that contain more particles than the number of levels in the Z dimension of the 3D LES data box are given a chunk of dynamically allocated shared memory to work with. Smaller kernels are not given any dynamically allocated shared memory. As in the brute force LES-LSM, the kernel begins by having each thread in a block load its particle location and wind information from global memory into shared memory buffers. Kernels with dynamic shared memory will use the first threads in a block to load the turbulent energy vertical profiles from global memory into the dynamically allocated memory space.

The kernel then enters a loop which concludes only when the particle's valid time has reached the end of the LES data on the device, or when a particle off-grid miss occurs. The first operations in the loop are related to finding the spacial and temporal locations of the particle. These operations are different from their brute force equivalents because tile boundaries must be considered, and the scalable kernel has more than 2 LES data-sets to manage. Aside from the extra book-keeping related spanning multiple LES data-sets and checking for off-grid misses, particle movement proceeds just as it did in the brute force implementation with the particle being translated forward in time based on the resolved and stochastic velocities. At the conclusion, all the shared memory particle information is written back to global memory.

6.1.4 Scalable Application Workflow

The workflow of the scalable GPU LES-LSM has departs significantly from the previous LES-LSM versions. Instead of having the strict algorithm structure which loops over the number of LES files, the number of release times, the number of release locations and the number of particles per release location, the scalable GPU LES-LSM algorithm is structured to pool all particles together in one set of arrays on the device. Important particle state information like release time and release location of a particle will be stored in the set of device arrays as well. Bringing all this information together on the device makes it possible to treat particles in terms of their current location in the LES volume, rather than their release location and time.

Figure 6.3 shows the workflow structure of the scalable GPU LES-LSM. At the highest level, the scalable model operates as a discrete event simulation. Given a range of time to model, the outer-most loop moves the simulation valid time, T_{sim}, forward until it reaches the final time T_{final}. Along the way, discrete events can occur which may add particles to the LES-LSM particle movement engine, or perform some other task. The only events in this current model are particle insertion events. The second **while** loop is responsible for moving T_{sim} forward to either the next event time, T_{event}, or to the final event time, T_{final}. T_{sim} is advanced by moving the simulated particles forward through either $((N_{times} - 1) * \Delta T_{LES})$ simulated seconds, or to the minimum of T_{final} and T_{event}. Particles are organized into work units and advanced forward by the combination of host thread 2 running the Work Request Manager preparing work and host thread 1 dispatching work to the device over CUDA streams. Finally, T_{sim} is updated to the minimum particle valid time of all the active particles.

Algorithm 5 Scalable GPU LES-LSM

```
Require: Range of LES files
Require: Particle release schedule (release times, release locations,
    particles per release)
Require: Scalable parameters $N_{times}, N_{workbuffers}$ etc...
 1: Allocate and prepare GPU arrays
 2: Create an event model tracking upcoming events
 3: while $T_{sim} < T_{final}$ do
 4:   Check for events at $T_{sim}$ and run events. (add particles)
 5:   Set $T_{event}$ to the next event time.
 6:   while $T_{sim} < MIN(T_{final}, T_{event})$ do
 7:     $T_{moveEnd} = MIN(T_{sim} + (N_{times} - 1) * \Delta T_{LES}, T_{final}, T_{event})$
 8:     GPU Pre: particle tiling on the device.
 9:
10:     Host Thread 1 -
11:     while Any work units tiles unprocessed. do
12:       if A CUDA Stream is available then
13:         Retrieve available work units from the work queue.
14:         Load work unit data to the device asynchronously over
            CUDA Stream.
15:         Move particles from time $T_{sim}$ to $T_{moveEnd}$ asyn-
            chronously on CUDA Stream.
16:       end if
17:       if A CUDA Stream completed work then
18:         Release the work unit assigned to that stream.
19:       end if
20:     end while
21:
22:     Host Thread 2 -
23:     while Any work unit tiles unprocessed. do
24:       Read the LES tile needed for this work unit.
25:       Add the work unit to the work queue for processing.
26:     end while
27:
28:     GPU Post: get the latest particle time, $T_{sim}$.
29:   end while
30: end while
31: Copy output grids from device.
32: Write model output.
33: Free device memory and release the device.
```

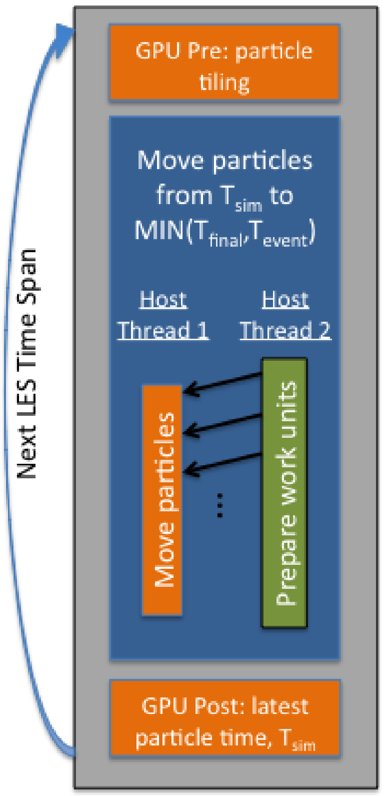

GPU Pre: particle tiling
Move particles from T_{sim} to $MIN(T_{final}, T_{event})$
Host Thread 1
Host Thread 2
Move particles
Prepare work units
Next LES Time Span
GPU Post: latest particle time, T_{sim}

Figure 6.3: Detailed textual workflow the scalable GPU LES-LSM along with condensed flow diagram. The flow diagram highlights the most time consuming operations including GPU particle tiling, GPU particle movement, GPU post processing, and work unit preparation. The flow diagram also illustrates the dependency between the work unit preparation host thread and the particle movement host thread.

6.2 Blocking Device Copies

It was observed that the asynchronous host to device copies used to copy work tiles to the device were not behaving asynchronously as expected. The problem seems to relate exclusively to the texture memory scalable GPU LES-LSM implementation. Figure 6.4 shows the same run of the LES-LSM model using a texture based particle movement kernel, and a texture free particle movement kernel (using linear memory) over the same simulated model increment. The blocks along the X axis, adjacent to the stream labels, represent time when a kernel is on the device, but not necessarily running. The texture free version exhibits significant streaming parallelism with each stream having a kernel on the device either preparing to run (copying data using cudaMemcpyAsync), or already running. The texture version, however, is blocking on the asynchronous copy of work to the device cudaArray (copying data using cudaMemcpyToArrayAsync) whenever a kernel is running on a different stream, thus serializing the device copies and kernel executions. The problem relates to the cudaMemcpyToArrayAsync not actually overlapping memory copies on one thread with kernel execution on other threads. Any overlap between streams in the figure is a result of logging problems due to the blocking copy operations on the host thread. Despite the lack of overlap, the texture kernels are at least twice as fast as the texture free kernels, and will be used throughout the remainder of the analyses.

The logging problems associated with using texture memory, and the cudaMemcpyToArrayAsync command, relate to the fact that the copy operations do behave asynchronously to the host (the cudaMemcpyToArrayAsync immediately returns control to the host thread) in cases when no kernel is running. In this case, our logging statements running on the host thread only know when the first asynchronous copy was issued on a given stream, and when that stream has completed the particle movement kernel. Timing information relating to the distinct copy and move operations has been confounded into one single time. Fortunately, the cuda profiler gives more information about the data transfer bandwidth, which can be used to statistically split the confounded time into memory transfer and particle movement components.

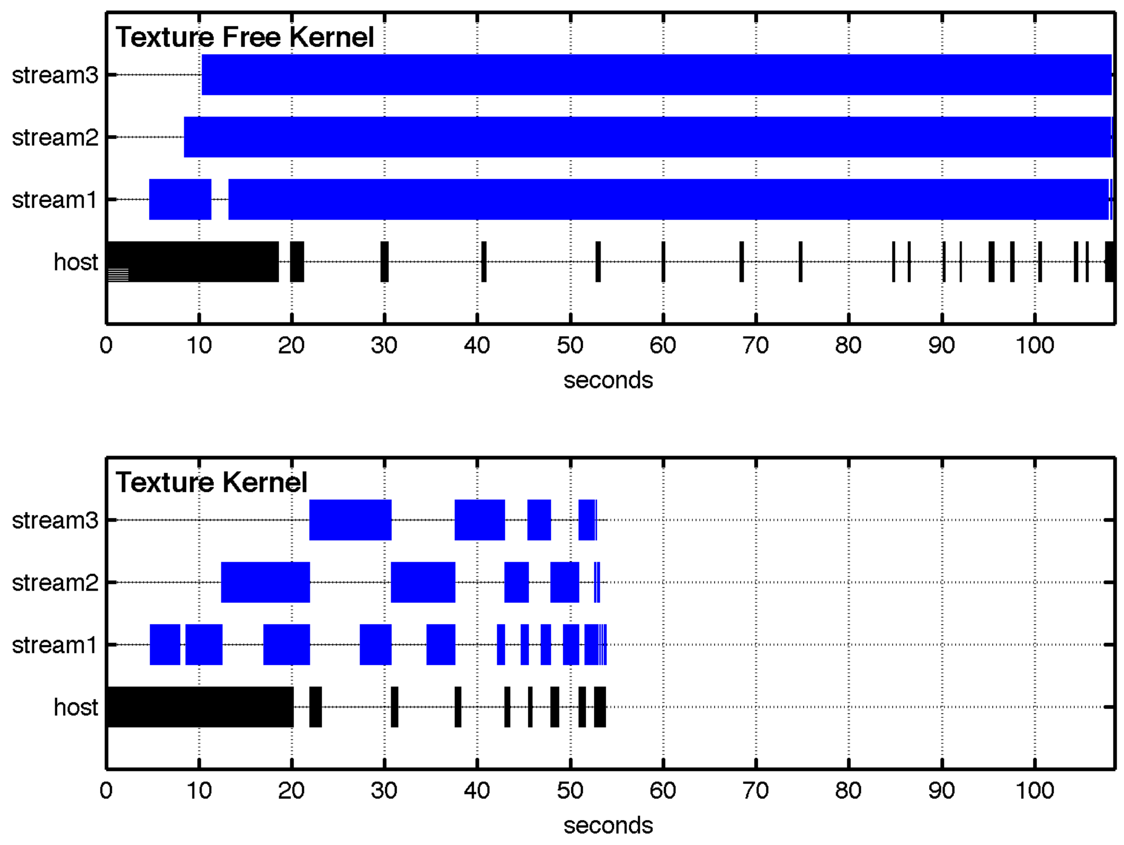


Figure 6.4: The two gantt charts show the time spent moving particles relative to the host on CUDA streams 1,2 and 3 for a texture based kernel and a texture free kernel. The texture free kernel shows the type of streaming parallelism expected when using asynchronous host to device memory transfers using CUDA streams, with each stream having a kernel executing on the device or waiting to execute on the device. The texture based kernel however is suffering from blocking of the host thread when running the host to device memory transfers that precede kernel execution. As a result, the texture based kernels and memory transfers are serialized. Any overlap on streams 1, 2 and 3 of the texture kernel is an artifact of the blocking memory transfers on the host thread.

Despite the fact that memory copies from host to device are not being overlapped by kernel execution, the CUDA streams based implementation was kept in the scalable GPU LES-LSM. The hope is that the streams based implementation may be applicable in later versions of CUDA where either cudaMemcpyToArrayAsync becomes able to properly overlap data transfer and kernel execution, or where multiple kernels can execute on a single GPU at once (as is possible with the "Fermi" generation of CUDA GPUs). Also, it may be possible to create a work around by transferring memory asynchronously to a region in linear memory, then copying that memory from linear memory to the cudaArray required for 3D texture usage.

6.3 Analysis

The scalable GPU LES-LSM introduces many new configuration options, with the two most closely related to performance being N_{times} and P_{tile}. These options directly affect the amount of sequential reading time required before the first kernel execution, and the total read time required to prepare the current work units for computation. When combined with information about the number of particles in flight, one can begin to understand how much of the time spent reading LES data is overlapped by GPU computation. The following sections analyze the performance of the scalable GPU LES-LSM across a range of configurations and problem sizes, and conclude by creating a performance model to represent the scalable GPU LES-LSM.

6.3.1 Parameter Selection

The first step in analyzing the performance of the scalable GPU LES-LSM is to isolate a set of values for N_{times} and P_{tile} value which perform well on the different release scenarios. This analysis was done using the test4a and test4s workloads because they were found to be computationally intensive enough to overlap 80 to 100% of the read operations (aside from the first tile read operation which is always sequential) with particle movement operations. The size of the tiling buffer zones were set to linearly increase in each dimension (x,y,z) with the number of N_{times} time values, thus leading to larger buffer zones with longer spans of time. The buffer zones were made sufficiently large so as to eliminate off-grid misses in all of the configurations.

Less influential parameters, such as $N_{streams}$ and $N_{workbuffers}$, were given default values that were selected manually. The the $N_{streams}$ value was set to 3, so as to provide at least two other streams for asynchronous memory transfer to the device while one of the streams is executing a kernel. The $N_{workbuffers}$ parameter was set to twice the size of $N_{streams}$ so that each stream will have one host buffer associated with a work unit being processed, and one host buffer associated with an upcoming work unit. It is important to keep $N_{workbuffers}$ reasonably small in order to limited the amount of pinned memory allocated on the host machine. The parameter values of $N_{streams} = 3$ and $N_{workbuffers} = 6$ were used in all the scalable GPU LES-LSM runs performed in this thesis.

Table 6.5(a) shows the scalable GPU LES-LSM total computation time for the test4a and test4s workloads across an N_{times} range of 3 to 6, and a P_{tile} range of 2 to 4 across the horizontal and vertical dimensions. The first observation about these data is that adding N_{times} intervals beyond four does not provide a computational benefit, as illustrated in Figure 6.5(b). This is mostly associated with the additional time needed to read the first work tile (sequential read) as the N_{times} increases. The few exceptions to the negative impact of increasing N_{times} values involve average workloads where the use of 4 or 5 time increments helps to better overlap the total amount of read time with particle movement computation.

A second observation about the performance data in Table 6.5(a) is that the spacial tiling parameters seem to affect the performance of the scalable GPU LES-LSM as a balance of two factors: the time to read the first work tile, and the total buffer zone needed to process all of the tiles. The time to read the first work tile decreases the most when tiles are smaller because there is less memory to be read. However, smaller tile sizes mean more total buffer zone needed to process all the tiles in the simulation (more 3D tile surface area to be buffered). Figure 6.5(c) shows the performance of the LES-LSM across the different horizontal P_{tile} decompositions. The single release scenario cases benefit the most from a 4x4 horizontal decomposition that decreases the time needed to read the first work tile. Average release scenarios, however, benefit the most from a 3x3 horizontal decomposition which is sufficiently small to reduce the time needed to read the first work tile, but sufficiently large to keep the GPU engaged and reduce the amount of buffer space needed. In all of the average and single release cases, the vertical tiling parameter of 4 provides the best performance for the same reasons as the horizontal dimensions.

The final configuration decided upon for single release scenarios was $N_{times} = 4$ and a $P_{tile} = 444$ to minimize the overall time needed to read the first work tile. The configuration for average releases was decided as $N_{times} = 4$ and $P_{tile} = 334$ because these options strike an even balance between minimal tile size and maximal computational work per tile.

6.3.2 Performance Results

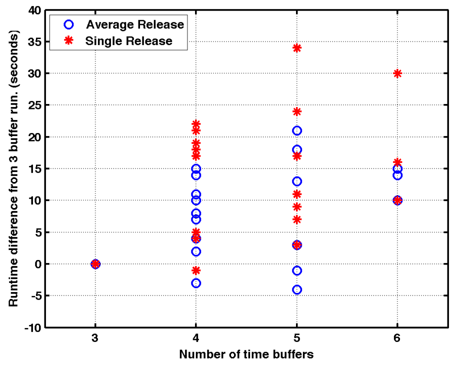
A summary of the scalable GPU LES-LSM's performance across the realistic workloads is shown in Table 6.1. This table has changed slightly from the sequential FORTRAN LES-LSM and brute force GPU LES-LSM tables because of the algorithm restructuring required for the scalable GPU LES-LSM. Specifically, the LES read and preprocess operation has been grouped into one item because the operations are difficult to separate. Also, the copy data to device operation is now being confounded with the the move particles operations, because of the blocking read operations. The total particle movement operation time was split into the its "move" and "load" components using the approximate total time needed to transfer data to the GPU. The approximate total transfer time was calculated by dividing the total bytes transferred to the device with the mean transfer bandwidth, 3.6476 (GB/s), observed in the scalable GPU LES-LSM using the CUDA profiler.

6.3.3 Performance Model Creation

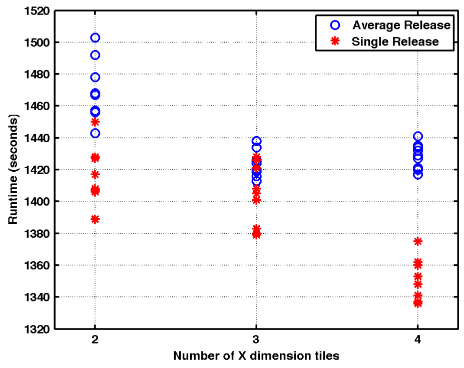
The scalable GPU LES-LSM performance model follows the same structure as Algorithm 5 with the exception that instead of having two host threads, the performance model has scheduling logic which attempts to mimic the interactions between the two threads. Performance time in the scalable GPU LES-LSM is a product of many factors, with the LES file reading and particle movement being the primary factors. The following models of the performance of the influential

		Number of Files			
Workload	Tiling	3	4	5	6
	2x2x2	1492	1503		
	2x2x3	1468	1478	1467	
	2x2x4	1443	1457	1456	1453
	3x3x2	1434	1438		
test4a	3x3x3	1424	1426	1420	1472
	3x3x4	1416	1413	1419	1466
	4x4x2	1420	1427	1441	
	4x4x3	1417	1432	1435	1432
	4x4x4	1421	1429	1434	1435
	2x2x2	1428	1450		
	2x2x3	1408	1427	1417	
	2x2x4	1389	1407	1406	1405
	3x3x2	1421	1426	1428	
test4s	3x3x3	1401	1405	1408	1488
	3x3x4	1380	1379	1383	1452
	4x4x2	1341	1362	1375	
	4x4x3	1336	1353	1360	1366
	4x4x4	1337	1341	1348	1347

(a) Scalable GPU LES-LSM Runtime Results (seconds)



(b) Time Buffer Influence (Fixed Tiling Configuration)



(c) Grid Configuration Influence

Figure 6.5: Exploration of configuration options for the scalable GPU LES-LSM.

		Workloads					
		ave1	ave2	ave3	sin1	sin2	sin3
Workload description	Total particles	162000	810000	1620000	162000	810000	1620000
Total time per operation	LES read/prep.	1507s	1019s	714s	1232s	784s	568s
	GPU pre	5.029s	5.665s	6.632s	5.313s	5.968s	7.368s
	GPU post	0.033s	0.159s	0.299s	0.033s	0.156s	0.300s
	Move particles*	323s	925s	1683s	311s	906s	1664s
	Move particles: move*	282s	881s	1638s	285s	880s	1639s
	Move particles: copy*	42s	44s	44s	25s	25s	26s
	Miscellaneous	4s	4s	4s	4s	4s	3s
	Total time	1839s	1954s	2408s	1552s	1700s	2243s
Percent of total time	LES read/prep	81.93%	52.16%	29.67%	79.38%	46.13%	25.34%
	GPU pre	0.27%	0.29%	0.28%	0.34%	0.35%	0.33%
	GPU post	0.00%	0.01%	0.01%	0.00%	0.01%	0.01%
	Move particles*	17.58%	47.33%	69.88%	20.02%	53.29%	74.21%
	Move particles: move*	15.45%	45.09%	68.24%	18.42%	51.82%	72.99%
	Move particles: load*	2.28%	2.27%	1.85%	1.64%	1.50%	1.16%
	Miscellaneous	0.21%	0.20%	0.16%	0.25%	0.22%	0.11%
FP statistics	Mean FP ops per byte	< 15.5	< 72.8	< 144.4	< 32.1	< 153.4	< 300.9
	Mean GFLOPS	6.954	12.265	13.678	7.761	12.864	13.989
	Mean GFLOPS % of peak	0.655%	1.155%	1.288%	0.731%	1.211%	1.317%

Table 6.1: Single precision FP computational performance results from the scalable GPU LES-LSM. GFLOPS statistics were calculated using the FP complexity observations from Section 3.3. The Mean GFLOPs % of peak metric uses a peak of 1062 GFLOPs for the GTX285 GPU. (* Move particles time includes the copy to device operation. The move particles time has also been split statistically using the memory transfer bandwidth.)

factors were derived using the results from the analyses in Section 6.3.1.

The time required to read data from the LES Data Server is modeled as two different linear functions on the number of bytes to be read, as shown in Figure 6.6. The structure of the LES data in memory means that the first read operations, which are generally focused on the lower vertical levels of the LES volume, scan over the higher vertical levels, thus loading them into host RAM. The two linear functions expressed in 6.6 attempt to capture that distinction. The line with the higher slope represents uncached reads of the lower LES levels which are more time consuming. The line with the lower slope represents cached reads of the upper LES levels. The correspondence between LES read heights and uncached or cached reads does not always hold, as can be seen from the stray observations in the scatter plot data. Despite the fact that there are some unrepresented factors at work, the LES read height distinction holds often enough to suit the performance model needs.

The compute time needed to move particles one simulated second in the scalable GPU LES-LSM particle movement kernel is modeled as a piecewise linear function of the number of particles. The model was derived from the combination particle movement times from the average and single release scenarios. The confounding of the particle movement statistics was removed by subtracting

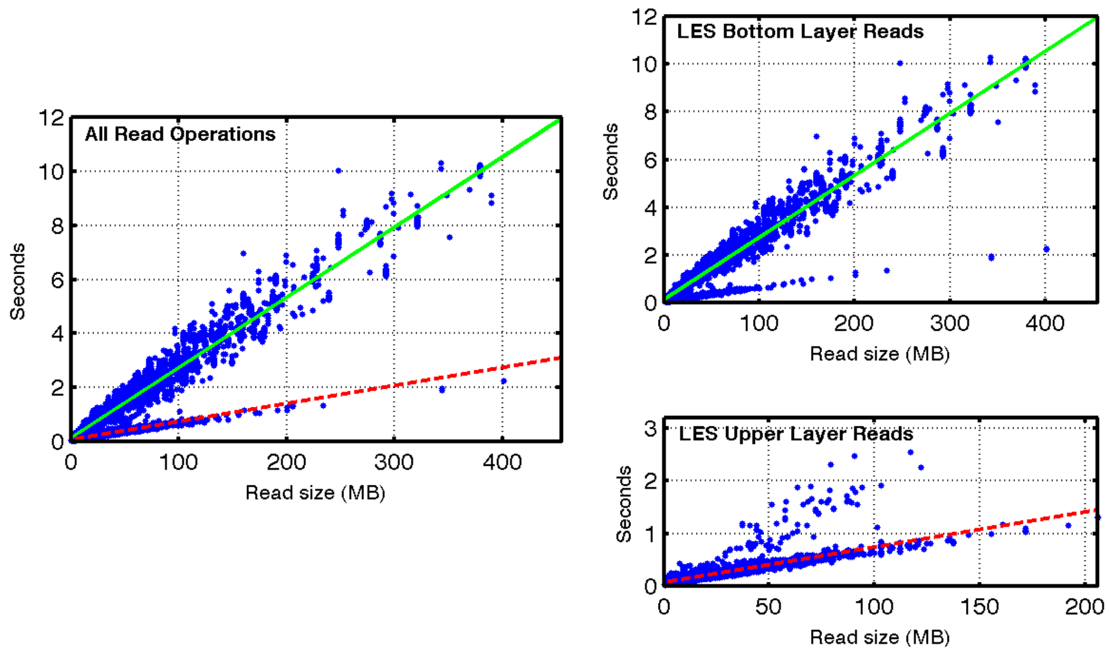


Figure 6.6: The performance time of LES data read operations is modeled by two linear functions. The solid line models reads of the bottom vertical layer of LES data tiles which are generally not yet present in system RAM. The dashed line models upper vertical layer LES data tile reads which are typically already cached in system RAM.

the copy to device time from each kernel invocation. The first line segment from 1 to 32,000 particles represents a regime where the GPU is not completely occupied. A kernel running 32,000 particles will have approximately 250 blocks of 128 particles in flight on the device, which gives each of the 30 multiprocessors approximately 8 blocks to schedule. The unsaturated regime plot of Figure 6.7 shows the measured kernel performance times for various kernel sizes from the range of 1 to 32,000. The obvious steps in the kernel performance data each represent when a kernel has N blocks for each multi-processor on the device. With 30 multi-processors, and 128 particles for block, these transitions occur every 3,840 particles until the kernels performance become too variable to distinguish them.

The dashed line in Figure 6.7 models the performance of kernels with the greater than 32,000 particles. There is a great deal of variability in the kernel performance at these large sizes due to factors like the variable model time-step and or the locality of particles within each warp of a kernel. Without fully understanding when and why this variability occurs, a linear fit seems to be the best model of performance, though it may tend to over-estimate kernel runtime for these large sizes.

Table 6.8 shows the scalable GPU LES-LSM performance model parameters. Variables which are not influenced significantly by the variability in factors are modeled as a mean and variance of the measured performance. The aforementioned linear models for data reading time and particle movement kernel time are described as functions of N_{bytes} and $N_{particles}$. The device copy operation is modeled using the mean memory transfer bandwidth (from pinned memory) observed using the CUDA profiler. Table 6.8(b) shows the observed and predicted performance of the scalable GPU LES-LSM across realistic workloads. As with the performance models for the sequential FORTRAN LES-LSM and the brute force GPU LES-LSM, the model tends to under predict the solution time for average release scenarios and over predict the solution time for single release scenarios (results shown in Table 6.8(b)).

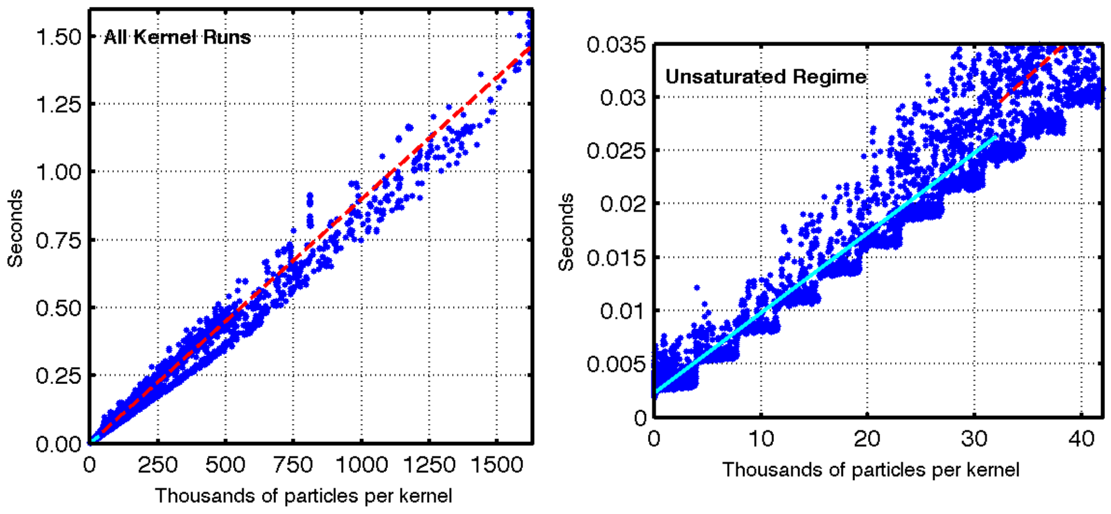


Figure 6.7: Measurements of the performance time needed for the particle movement kernel to move particles one simulated second. The solid line models the unsaturated regime of particles counts 1 through 32,000, while the dashed line models particle counts greater than 32,000.

Operation	Time Mean (sec)	Time Variance (sec^2)	# Samples
Per run: startup	1.3874	0.0053	61
Per run: closing	0.9110	$4.2075*10^{-4}$	61
GPU Pre: particle tiling	0.1004	$1.2591*10^{-3}$	1918
GPU Post: check latest time	0.0082	$1.7557*10^{-6}$	1918
	Linear Models		Bandwidth
movement kernel($N_{particles} < 32000$)	$7.5061 * 10^{-7}(sec/part) * N_{particles} + 0.0023(sec)$		$1.3322*10^6$ (part/sec)
movement kernel($N_{particles} >= 32000$)	$8.9858 * 10^{-7}(sec/part) * N_{particles} + 0.0005(sec)$		$1.1129*10^6$ (part/sec)
data read (lowest vertical level)	$2.5917 * 10^{-8}(sec/byte) * N_{bytes} + 0.1288(sec)$		38.585 (MB/s)
data read (upper vertical level)	$6.6671 * 10^{-9}(sec/byte) * N_{bytes} + 0.0813(sec)$		150.0 (MB/s)
copy to device	$2.7415e * 10^{-10}$(sec/byte) $*N_{bytes}$		3.6476 (GB/s)

(a) Scalable GPU LES-LSM Performance Model Parameters

Workload:	ave1	ave2	ave3	sin1	sin2	sin3
Observed (sec)	1823	1953	2401	1549	1699	2245
Predicted (sec)	1775	1928	2239	1668	1885	2325

(b) Scalable GPU LES-LSM Performance Model Error

Figure 6.8: Scalable GPU LES-LSM Performance Model Parameters and Validation

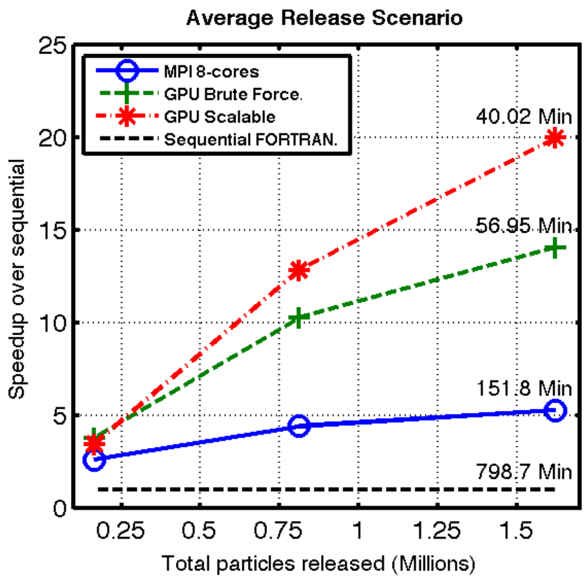
Chapter 7

Performance Comparison

The performance of the LES-LSM implementations were compared using total application speedup over the single-core sequential implementation in Figure 7.1. The Average Release Scenario line graph shows the results of the realistic workloads with average release scenarios, and the Single Release Scenario graphs show the results of the realistic workloads with single release scenarios. The X-axis of each graph represents the number of particles in each simulation, with the symbols along each line indicating the three workloads in numeric order from left to right.

For the smaller workloads (ave1 and sin1), where the sequential FORTRAN LES-LSM is read bound, the GPU implementations provides little benefit when compared with a multi-core CPU approach. As the workloads increase in size, and the amount of particle movement time in the sequential FORTRAN LES-LSM increases, the true benefits of the GPU approach are realized. In the largest workloads (ave3 and sin3), the brute force and scalable implementations observe application speedups of approximately 14x and 20x, relative to the sequential FORTRAN implementation. Compared against the unsophisticated (and unoptimized) MPI implementation, the brute force and scalable implementations observe approximately 2.6x and 3.8x speedups, respectively. According to these results, to achieve the the same performance as the GPU implementations in a multi-core CPU framework would require a 3 or 4 node cluster of our 8-core host machine.

The performance of the two GPU implementations also differs across the range of observed problem sizes. The brute force and scalable GPU implementations perform approximately the same for small workloads (ave1 and sin1), with the scalable version actually performing worse in the ave1



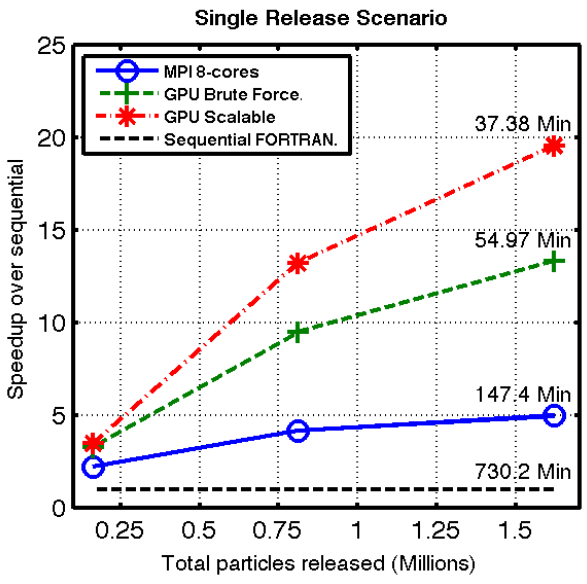


Figure 7.1: The speedup over single-core sequential implementation results of various LES-LSM implementations across multiple workloads. The left plot shows speedup for the average release workloads, ave1, ave2 and ave3, ordered left to right and translated to total number of particles. The right plot shows speedup for the single release workloads, sin1, sin2 and sin3, ordered left to right and translated to total number of particles.

workload. As workload sizes increase, the scalable implementation begins to outperform the brute force implementation. The improved performance in the scalable implementation is a result of changes to the LES read pattern, changes to how memory is transfered to the GPU, and changes to the way the particle movement kernel is structured. The following sections describe each of these differences and how they affect the speedup values shown in Figure 7.1.

7.1 LES Data Read and Preprocess

The LES data read operation is very time consuming and can quickly become the primary computational bottleneck of the LES-LSM when particle counts are low, or when the particle movement operation has been accelerated by a GPU. The upper chart in Figure 7.2 shows the total number of LES bytes read from disk over each of the realistic workloads. The lower chart shows how much time was spent reading and preprocessing LES data sequential, without being overlapped by particle movement computation. The brute force implementation reads data in the same pattern as the sequential FORTRAN implementation, but takes more time because the brute force read and preprocess operation is less efficient (implemented in C++). The scalable implementation reads data as a series of tiles, which reduces the total amount of LES data read from disk. However, there are computational overhead costs associated with reading the LES data as a series of tiles (illustrated by the "GPU Scalable: All" bars of Figure 7.2). Fortunately, the scalable implementation is able to overlap some of the LES read and processing time with particle movement operations on the GPU (sequential reads, which are not overlapped, shown in the "GPU Scalable: Sequential" bars of Figure 7.2). On smaller workloads (ave1 and sin1), the scalable implementation's read/preprocess operations behave very poorly because they are not well overlapped by particle movement operations. As workloads increase in size, more of the LES read/preprocess operations are overlapped by particle movement computation, thereby reducing the total amount of sequential read/preprocess time significantly.

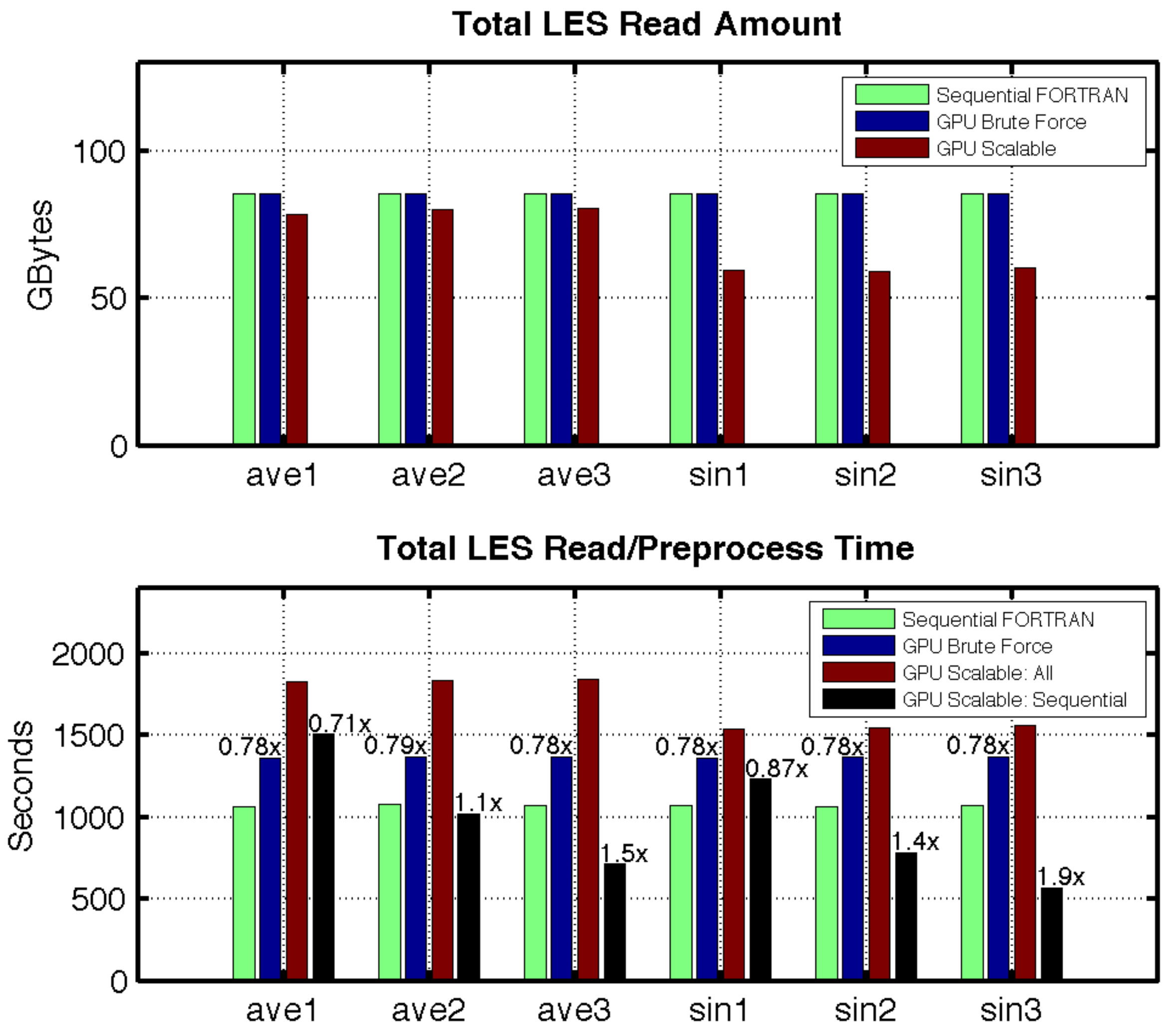


Figure 7.2: The amount of LES data read from disk and the amount of sequential (not overlapped by computation) time taken to read and preprocess that data across the realistic workloads. Embedded speedup values represent the speedup of the LES read and preprocess operation relative to the sequential FORTRAN read and preprocess operation.

7.2 Memory Transfer to Device

LES data must be communicated to the GPU for particle movement computation to occur. That communication comes at a temporal cost of transferring data over the system bus. In both of the GPU implementations of the LES-LSM, the GPU transfers are being performed synchronously, and are not being overlapped by computation on the GPU. The upper chart in Figure 7.3 shows the number of bytes sent to the GPU by the brute force and scalable implementations, across the realistic workloads. The lower chart shows the number of seconds spent transferring that data to the GPU. Because the scalable implementation uses the tiled work unit paradigm, it has to transfer much more data to the GPU. However, the scalable implementation's transfers are faster because they use pinned memory on the host (3.6476 GB/s) while the brute force implementation use linear memory (2.1786 GB/s). In any case, the differences in memory transfer cost between the brute force and scalable implementations are dwarfed by those of LES data read and particle movement.

7.3 Particle Movement Bandwidth

The brute force and scalable GPU implementations use CUDA kernels to move particles through simulated time. Figure 7.4 shows the performance model derived computational speedup of the brute force and scalable implementations across very large workloads where particle movement calculations are the primary computational factor. As the workloads continue to increase in size, the total application speedups approach the particle movement bandwidth speedups shown in Table 7.4(b). Fundamentally, the brute force and scalable kernels are the same, with the exception that the scalable kernel has to handle tile indexing and boundaries, and that the scalable kernel carries particles through longer spans of simulated time. Computationally, the two biggest differences between the brute force and scalable kernels are the number of times which they are launched, and the locality of particles within the blocks of the kernels.

The number of kernel launches is important because each kernel launch incurs a certain amount of overhead. As an example, in the middle of the ave3 workload, the brute force kernel

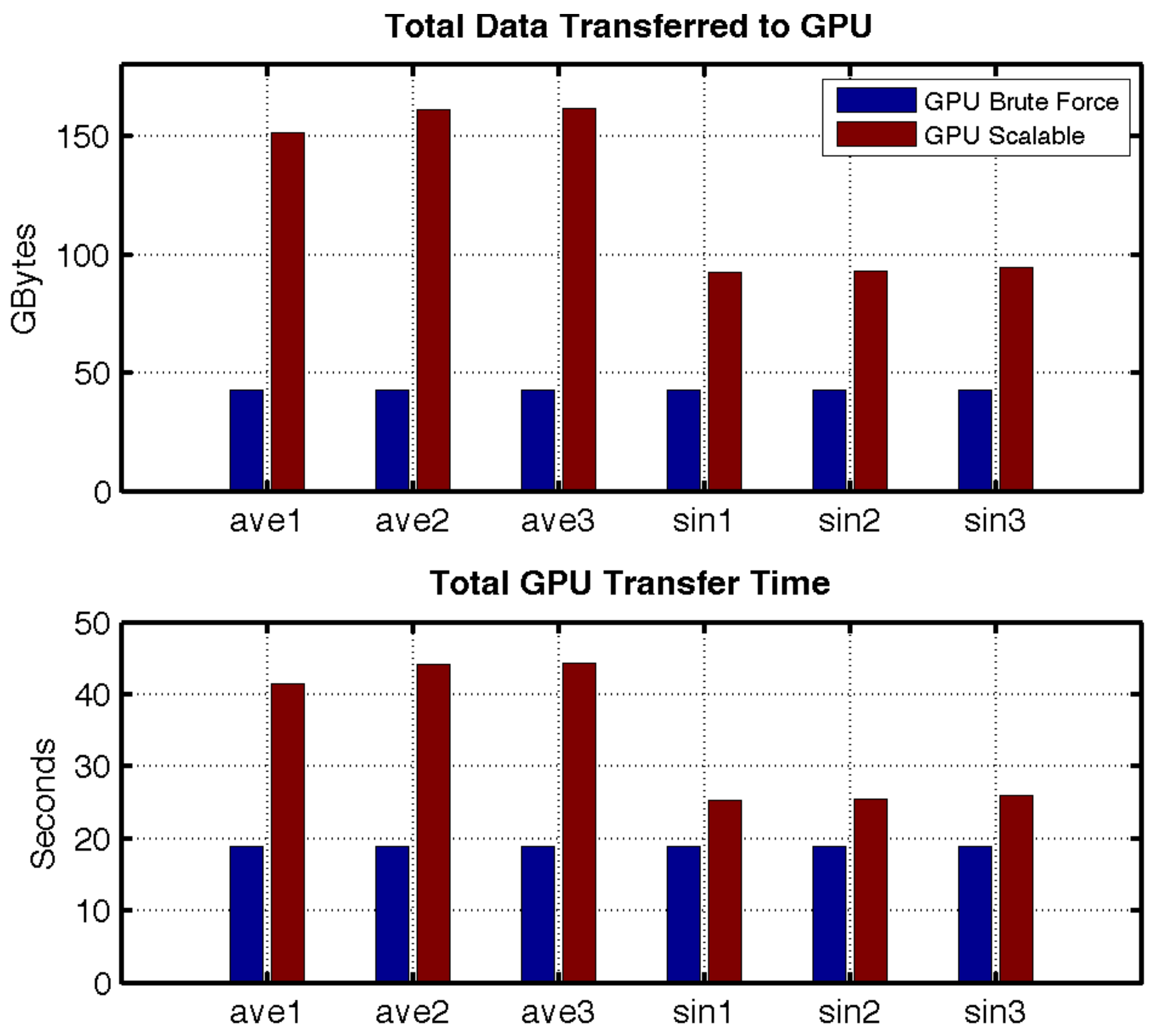


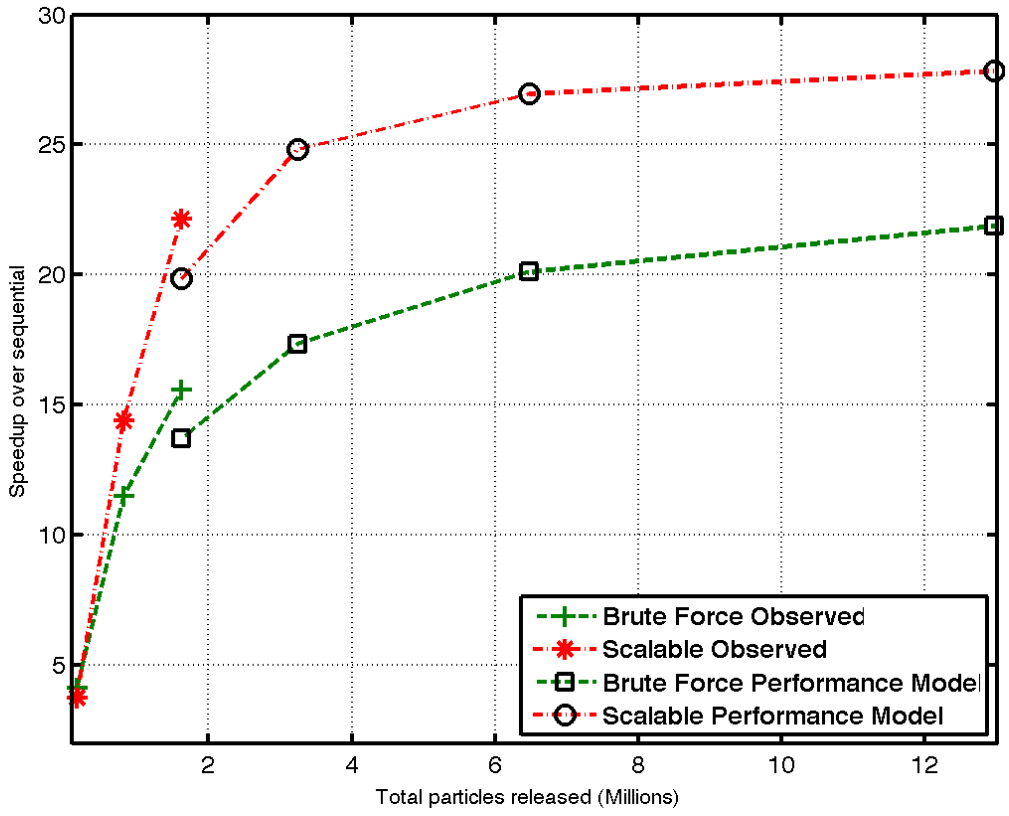
Figure 7.3: The cumulative amount of LES data transferred to the GPU and the total transfer time observed across the realistic workloads. Linear memory transfers were used in the brute force GPU LES-LSM and pinned memory transfers were used in the scalable GPU LES-LSM.

would require 2400 kernel launches to move particles through 3 LES times, while the scalable version would only require 36 launches. With each launch incurring approximately 10 to 100 micro-seconds of overhead, the difference between 2400 launches and 36 launches is approximately 0.2 seconds (assuming a worst case scenario of 100 micro-second overhead). This may not be a first-order problem, but it highlights the improved efficiency of the scalable implementations in spanning multiple LES times.

Particle locality within the blocks of a kernel is important to the performance of the brute force and scalable implementations because CUDA texture caches are only shared across the threads of a block (per multiprocessor). To make full use of the texture cache, it is best to have all the particles in a block accessing the same sub-region of the LES volume. Unfortunately for the brute force implementation, particle locality is only guaranteed in the time period following the release of particles. Within 15 or 20 LES time steps, significant spread has occurred, and the particles that are adjacent in each block are no-longer within the caching range of one another. The tiling operation of the scalable LES-LSM helps achieve better performance simply by organizing particles together in the same tile, thus increasing the likelihood of texture cache hits for particles within that tile. The increased particle locality of the scalable implementation is the main factor behind its higher particle movement bandwidth.

In further illustration that particle locality is driving the increased performance of the scalable implementation, a brief study was conducted to further increase particle locality by sorting particles within each work unit tile. To keep this exercise as simple as possible, the same tiling technique used in the scalable GPU LES-LSM to create 3D tiles was used to create very small sub-tiles (on the scale of LES 12 x 12 x 6 pixels) for grouping particles within a computational 3D work unit tile. Sub-tiles were ordered along the Y-axis of the data to better fit the mean wind of the scenario, thus making along-wind particle sub-tiles adjacent in memory and more likely to be computed in the same block or warp.

Figure 7.5 shows the particle movement kernel timing results of the normal scalable GPU LES-LSM and the enhanced locality scalable GPU LES-LSM. The average release scenario work-



(a) Modeled GPU Performance For Large Workloads

Particle Movement Bandwdith		
Version	Bandwidth (part/sec)	Speedup
Sequential FORTRAN	34,960	1x
GPU Brute Force	850,020	24.3x
GPU Scalable	1,112,900	31.8x

(b) Particle Movement Bandwidth

Figure 7.4: The observed speedup for the ave1,ave2 and ave3 workloads, and theoretical speedups for larger workloads. Theoretical speedups calculated using the algorithm performance models.

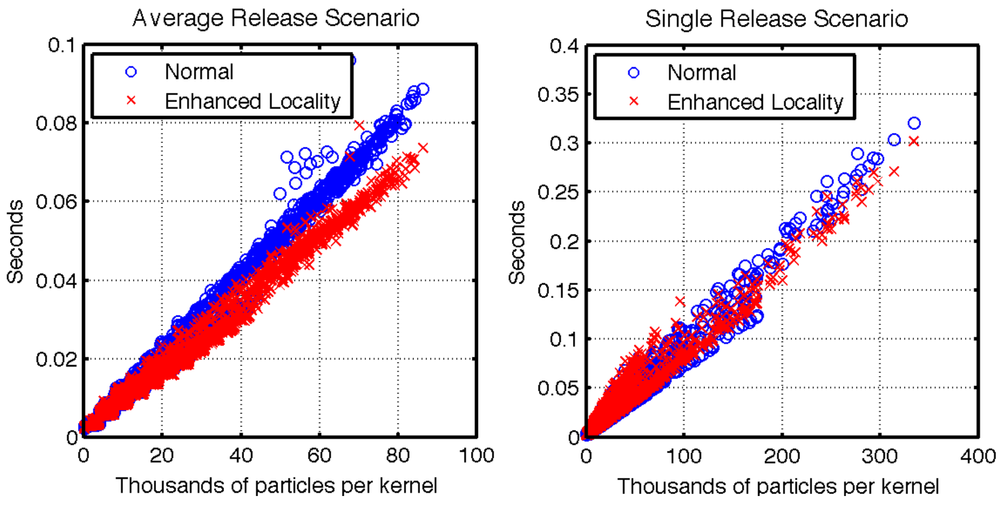


Figure 7.5: The kernel execution time of the scalable LES-LSM and the particle locality enhanced scalable LES-LSM run across the ave3 and sin3 workloads (lower is better).

loads benefit substantially from the increased particle locality brought about by the sub-tile ordering of particles, going from a particle movement bandwidth of 1,026,098 particles/sec to one of 1,236,796 particles/sec (bandwidth calculated from the least squares linear regression fit). The single release scenario exhibits a mixed response to the sub-tile ordering with large particle counts showing a clear benefit to the sub-tile ordering, while smaller particle counts show worse performance in some cases. The poorly performing kernels are likely a result of the particle sorting technique splitting particles across the sub-tile boundaries in a way that reduces the overall locality of that particular tile. The sin3 particle movement bandwidth of the normal scalable GPU LES-LSM and the enhanced locality version are 1,104,281 particles/sec and 1,158,031 particles/sec, respectively.

Chapter 8

Conclusions

The LES-LSM is an interesting problem for GPU acceleration because it stretches the limits of the GPU architecture. With high demands for LES data, the LES-LSM challenges the GPU architecture through expensive LES read operations, large memory transfers to the device, and abundant non-localized global memory accesses into the LES volume. The LES-LSM also requires a complex kernel with many registers assigned to each GPU thread, thus limiting the total number of active threads on a device. Despite these limitations, the brute force GPU LES-LSM, representing a shortest route to GPU parallelism, observed speedups of approximately 15x on large workloads. Further performance improvements were realized in the scalable GPU LES-LSM which restructures the LES-LSM problem to better suit the GPU co-processor paradigm.

The tiled work decomposition strategy used in the scalable GPU LES-LSM reduces the temporal cost of the LES read operation by overlapping them with GPU computation. In larger workloads, over 50% of the total tiled LES read time was able to be overlapped by particle movement computation. Reading LES data as tiles also reduced the total amount of LES data that needed to be read, particularly in the single release scenario workloads. In total, the scalable GPU LES-LSM is able to speedup the LES read and preprocess section by 1.5x and 1.9x over the sequential FORTRAN LES-LSM for average and single release scenario workloads.

The scalable GPU LES-LSM implementation attempts to reduce the temporal costs of transferring data to the GPU by overlapping data transfer with kernel execution using CUDA streams. Unfortunately, the cudaMemcpyToArrayAsync function used to copy data from the host to a cudaArray (required of 3D textures) on the device did not allow the overlap we had anticipated. The stream based structure of the scalable GPU LES-LSM was preserved in case the capabilities of the cudaMemcpyToArrayAsync function change with later version of CUDA hardware and software, or in case a workaround is discovered.

The tiled work decomposition strategy helps increase the particle movement bandwidth of the scalable GPU LES-LSM in two ways. First, by allowing particles to be carried through multiple LES volumes in a single kernel invocation, the total number of kernel invocations is reduced relative to the brute force GPU LES-LSM which only carries a kernel for δt_{save} simulated seconds. Reducing the number of kernel invocations reduces the total overhead associated with launching kernels, and leads to a more efficient implementation. Secondly, grouping particles into 3D work unit tiles significantly increases particle locality within the blocks of a scalable GPU LES-LSM kernel invocation, thereby increasing texture cache hit probability within that kernel. These two improvements explain why the scalable GPU LES-LSM is able to achieve particle movement bandwidths of 1,112,900 particles/sec while the brute force GPU LES-LSM only achieves 850,020 particles/sec.

Overall, the scalable GPU LES-LSM reliably outperforms the brute force GPU LES-LSM on larger workloads (greater than 750k particles in total). Maximum speedups of 20x and 15x over the sequential FORTRAN LES-LSM were observed, respectively, by the scalable GPU LES-LSM and the brute force GPU LES-LSM. Projecting forward to ever larger workloads using our computational performance models, the scalable and brute force implementations approach speedups of 31.8x and 24.3x (the observed particle movement bandwidth speedup) over the sequential GPU LES-LSM.

Chapter 9

Future Work

There are several parts of the LES-LSM which could benefit greatly from detailed performance examination. The LES read operation performed by the LES-LSM is one which requires significant improvement if the LES-LSM is ever to be parallelized further. The most obvious improvement to the LES read operation is to convert the double precision LES data files to single precious format, thus cutting the data size in half. This conversion should reduce LES read cost by at least half. Another potential improvement to the LES read operation would be to restructure the LES data file's internal structure to better suit a tiled access pattern. Finally, the C/C++ preprocessing step is in need of analysis to determine why it is so much slower than its FORTRAN counterpart. These improvements will make the LES-LSM better suited for future GPU and CPU parallelism.

A version of LES-LSM is currently being adapted to create instantaneous realizations of the particle field, thus moving the LES-LSM from prototype to production. This production version will likely be analyzed and adapted to the GPU using the tools written for this thesis. If larger workload sizes are expected in the new model, a multiple GPU implementation may be considered for additional performance. One second order feature which may be considered in future enhancements is a dynamic tiling algorithm which attempts to cluster particles by location in the LES domain and minimize the total number of tiles used for a given particle distribution.

Any future LES-LSM comparisons between GPU and CPU implementations will place a high priority on achieving multi-core CPU performance through algorithm restructuring and exhaustive compiler analysis. Optimizing the LES-LSM for both GPU and CPU platforms will provide a better understanding of the true benefit of a GPU architecture, relative to the increased development cost (and other negative factors) of utilizing a GPU architecture. A recent analysis by Intel Corporation [6] examined several benchmark applications that were highly optimized for both an Intel multi-core CPU (Core i7 960 processor) and an NVIDIA GPU (GTX 280). The Intel group observed a mere 2.5x speedup of the GPU platform over the CPU platform on average, thereby questioning the true benefit of GPU technology for scientific computing. Optimizing the future CPU and GPU versions of the LES-LSM to their fullest will ensure that our performance results are meaningful to the scientific computing community.

Bibliography

[1] N. Goedel, T. Warburton, and M. Clemens. Gpu accelerated discontinuous galerkin fem for electromagnetic radio frequency problems. In Antennas and Propagation Society International Symposium, 2009. APSURSI '09. IEEE, pages 1 –4, 1-5 2009.

[2] M. Harris. Fast fluid dynamics simulation on the gpu. In SIGGRAPH '05: ACM SIGGRAPH 2005 Courses, page 220, New York, NY, USA, 2005. ACM.

[3] Intel Corporation. Intel xeon processor - intel microprocessor export compliance metrics. Website, March 2010. http://www.intel.com/support/processors/xeon/sb/CS-020863.htm.

[4] M. Januszewski and M. Kostur. Accelerating numerical solution of stochastic differential equations with cuda. Computer Physics Communications, 181(1):183 – 188, 2010.

[5] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. Website, 2010. http://www.khronos.org/opencl/.

[6] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. SIGARCH Comput. Archit. News, 38(3):451–460, 2010.

[7] W. Li, Z. Fan, X. Wei, and A. Kaufman. Gpu-based flow simulation with complex boundaries. In GPU Gems 2, pages 747–764. Addison-Wesley, Boston, MA, 2005.

[8] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu. Multi-core acceleration of chemical kinetics for simulation and prediction. In SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pages 1–11, New York, NY, USA, 2009. ACM.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 190–200, New York, NY, USA, 2005. ACM.

[10] J. Michalakes and M. Vachharajani. Gpu acceleration of numerical weather prediction. Parallel Processing Letters, 18(4):531–548, 2008.

[11] F. Molnar, Jr., T. Szakaly, R. Meszaros, and I. Lagzi. Air pollution modelling using a graphics processing unit with cuda. Computer Physics Communications, 181(1):105 – 112, 2010.

[12] NVIDIA. NVIDIA CUDA C Programming Best Practices Guide, cuda toolkit 2.3 edition, July 2009.

[13] NVIDIA. NVIDIA CUDA: Programming Guide, version 2.3 edition, July 2009.

[14] NVIDIA. Whitepaper - nvidia's next generation cuda compute architecture: Fermi. Technical report, 2009.

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. Proceedings of the IEEE, 96(5):879–899, 2008.

[16] P. S. Pacheco. Parallel programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[17] E. R. Pardyjak, B. Singh, A. Norgren, and P. Willemsen. Using video gaming technology to achieve low-cost speed up of emergency response urban dispersion simulations. In Seventh Symposium on the Urban Environment. University of Utah Salt Lake City and University of Minnesota Duluth, September 2007.

[18] J. M. P. Piotr K. Smolarkiewicz. Towards mesh adaptivity for geophysical turbulence: continuous mapping approach. International Journal for Numerical Methods in Fluids, 47(8-9):789–801, 2005.

[19] The Portland Group. CUDA Fortran: Programming Guide and Reference, v1.0 edition, November 2009.

[20] P. Pospichal, J. Jaros, and J. Schwarz. Parallel genetic algorithm on the cuda architecture. Applications of Evolutionary Computation, pages 442–451, 2010.

[21] D. Roberti, R. Souto, H. de Campos Velho, G. Degrazia, and D. Anfossi. Parallel implementation of a lagrangian stochastic model for pollution dispersion. pages 142 – 149, oct. 2004.

[22] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 73–82, New York, NY, USA, 2008. ACM.

[23] I. Senocak, J. Thibault, and M. Caylor. Rapid-response urban cfd simulations using a gpu computing paradigm on desktop supercomputers. In Eighth Symposium on the Urban Environment. Boise State University, Boise, ID, January 2009.

[24] J. W. Sheaffer, D. P. Luebke, and K. Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pages 55–64, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[25] R. I. Sykes, C. P. Cerasoli, and D. S. Henn. The representation of dynamic flow effects in a lagrangian puff dispersion model. Journal of Hazardous Materials, 64(3):223 – 247, 1999.

[26] D. J. Thomson. Criteria for the selection of stochastic models of particle trajectories in turbulent flows. Journal of Fluid Mechanics Digital Archive, 180(-1):529–556, 1987.

[27] A. Walsh. Tokyo tech weather forecasting model gets 80x perf boost through gpu acceleration.

[28] J. C. Weil, P. P. Sullivan, and C.-H. Moeng. The use of large-eddy simulations in lagrangian particle dispersion models. Journal of the Atmospheric Sciences, 61(23):2877–2887, 2004.

[29] P. Willemsen, A. Norgren, B. Singh, and E. Pardyjak. Development of a new methodology for improving urban fast response lagrangian dispersion simulation via parallelism on the graphics processing unit. In Proceedings of the 11th International Conference on Harmonisation within Atmospheric Dispersion Modelling for Regulatory Purposes. Queen's College, University of Cambridge, United Kingdom, 2007.