CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–11
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Performance Study on CUDA GPUs for Parallelizing the Local Ensemble Transformed Kalman Filter Algorithm

Timothy Blattner*and Shiming Yang

Department of Computer Science, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250

SUMMARY

Modern graphics cards provide computational capabilities that exceed current CPUs. As one of the computational intensive problems, numerical weather prediction (NWP) has the opportunity to benefit from the massive number of threads and large memory throughput in the graphics architecture. In this paper, we present the key steps to integrate the CUDA programming framework for one key component in NWP, the data assimilation algorithm, which incorporates the observational data into the model to produce the best initial condition in the next prediction. The data assimilation algorithm we studied in this paper exhibits good localization and favors parallelism. To maximize the throughput of the graphics card, over a million CUDA threads, global memory coalescing, and fast graphics shared memory are utilized. We also demonstrate the differences in the advancement of GPU architectures from the GTX 200 series to Fermi. The experiments are carried out separately on a GTX 260 (GTX 200 series) and a GTX 460 (Fermi) graphics card. Results show an improvement of $72.1\times$ speed-up running on the GTX 260 and $92.7\times$ speed-up on the GTX 460. The results provide attractive evidence for applying CUDA GPUs to high demanding scientific computation realms. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: GPU, data assimilation, performance study, throughput, speedup

1. INTRODUCTION

With the uprising of multi and many core architectures, large scale problems are embracing GPUs for massive thread level parallelism. In the top 5 supercomputers, three of them deployed GPUs to achieve massive computing power. Amongst them, the top 1 Tianhe-A supercomputer (Top500 Supercomputer list November 2010 [3]) equips 7168 NVIDIA Tesla M2050 GPUs, while the top 3 Nebulae supercomputer [3] installs 4640 NVidia Tesla C2050 GPUs. To meet the demands of the platforms, increased memory, higher precision, and ECC have been integrated. To best utilize the computing capabilities provided by the graphic processors, it is highly desired to study how to optimize algorithms and programs on them. Nowadays, we consider the parallelism not only among the nodes in a cluster, but also on the microprocessor level by virtue of multicores and manycores. To fully utilize large clusters, a hybrid process and thread level parallel implementation can be adopted to accelerate the large scale problems.

Programming on GPUs and on CPUs are quite different. Unlike CPUs, which are designed with sophisticated branch prediction and enhanced instruction pipeline, GPUs are designed to handle graphics rendering of large proportion of data parallelism [8]. However, GPUs should not be confined to video game acceleration, but can be extended for general computation. In this paper,

*Correspondence to: Department of Computer Science, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250. Email: tblatt1@umbc.edu

we demonstrate one example of scientific computation, the local ensemble transform Kalman filter (LETKF) algorithm for data assimilation. We study how well it fits into the GPU architecture, by analyzing the algorithm and comparing the performance of two implementations on CPU and CUDA, respectively.

Data assimilation is a technique widely used to incorporate observational data into a prediction model, and provide initial conditions to the model for improved predictions [9, 10]. Data assimilation algorithms and their implementations are computationally intensive tasks [11], and require both accuracy and efficiency. The National Weather Service (NWS) in the U.S. performs predictions every 6 hours with the 3D-VAR algorithm. The European Center of Medium-range Weather Forecast (ECMWF) currently uses the world's finest resolution at about 25 kilometers, and offers a 10-day forecast using the 4D-VAR algorithm. Due to the scale of the forecasting area and the number of state variables used to describe the atmosphere and weather, the weather prediction problem is of terabyte scale and should be done in near real-time. For example, suppose we wish to perform a very refined numerical weather prediction for the entire U.S. continent at 1 kilometer horizontal resolution, 100 vertical levels 10000 meters above the ground, and there are 200 interesting physical variables. The basic data required for merely the state vectors of this problem are at least 3 terabytes. Nowadays, many data assimilation applications involve the ensemble Kalman filter (EnKF) methods, which have shown to be effective and easier for implementation.

The LETKF is proposed by Hunt et.al.[4], which exhibits efficiency and favors parallelism for data assimilation with very large non-linear dynamic systems in both sparse and dense data regimes[4]. Some studies exert efforts on parallelizing the implementation of the LETKF on clusters with MPI [5, 6]. This solution divides the main domain into sub-domains, and distributes each small problem to many computational nodes. Each node has one CPU, and separate memory. With a sufficient number of nodes, we can minimize the problem size on each node. Nevertheless, with this design there are a number of other issues. Firstly, using more nodes means higher costs on cluster building and power consumption. Secondly, the inter-node communication may increase significantly when the problem is sliced into very small sizes. Therefore, to release this bottleneck, many-core parallelism can be adopted for problems that are highly parallel and require only local information.

In this paper, we do not seek to compare the performance of our implementation on the CPU with that on the GPU, but will focus on the analysis of the data parallelism in a portion of the LETKF algorithm and how it fits into the GPU architecture. This paper is organized as follows. In section 2, we provide notations used throughout this paper, and analyze the LETKF algorithm for its data parallelism. Next, we specify that memory hierarchy and massive thread spawning in the GPU architecture are beneficial to the localization in the LETKF algorithm. In section 3, we report the speedup result obtained from different GPU optimization strategies. Lastly, we discuss future study.

2. PROBLEM ANALYSIS

The data assimilation method is used to recursively provide the initial conditions that the model could best forecast in short-range system states [10]. The process of data assimilation is made up of the following ingredients: the model, the observational data and the assimilation algorithm. In our experiments, we adopt the shallow water equation [7] as a model problem. This model problem simulates random water drops and the propagation of disturbances in a finite shallow water area. Given a problem domain such as an $n \times n$ evenly spaced mesh grid, the state vector is of size n^2. Assuming that every grid point has observational data, the observational vector is also of size n^2. However, in real cases, the dimension of observation is far less than the state vector. In our model problem, the model state $\mathbf{x}_t$ represents the height of a wave at time t. The wave is then projected into the observational space as $\mathbf{y}_t$, where measurement instruments can be used to observe some interesting physical quantities, such as the wave height. Within the system state and observations there contain model and measurement error, which are represented by random variables. Therefore, we need to know about the probability distribution of the state variables. Due to the difficulty of

DOI: 10.1002/cpe

tracking the distribution [4], the ensemble Kalman filter based methods use an ensemble of states $\mathbf{x}^{(i)}$ to simulate the distribution, where $i = 1, 2, \ldots, k$, and k is the number of ensemble members. Let $\mathcal{H}$ denote the projection operator, and $\mathbf{y}^o$ denote the measured wave height. The shallow water model, denoted as $\mathcal{M}$ forwards the system states from time step t to the next time step $t+1$ by

$$\mathbf{x}^i_{t+1} = \mathcal{M}(\mathbf{x}^i_t) + \varepsilon_{t+1}. \tag{1}$$

In the LETKF algorithm, we use the background ensemble $\mathbf{x}^{b(i)}_t$ as one of the inputs, which is forwarded by the model from the last time step's analysis ensemble. We also use the observation ensemble $\mathbf{y}^{b(i)}_t$, which is a projection of the background ensemble. Let k be the number of members in the ensemble. Also, the means and covariances of the members are required in calculating the new analysis ensemble [4]. The following equations illustrate the loop of finding $\mathbf{x}^{a(i)}_t$.

$$\mathbf{x}^{b(i)}_t = \mathcal{M}(\mathbf{x}^{a(i)}_{t-1}) \tag{2}$$

$$\mathbf{y}^{b(i)}_t = \mathcal{H}(\mathbf{x}^{b(i)}_t) \tag{3}$$

$$\mathbf{x}^{a(i)}_t = \text{analysis with } \mathbf{x}^{b(i)}_t, \mathbf{y}^{b(i)}_t, \mathbf{y}^o. \tag{4}$$

2.1. *Local Ensemble Transform Kalman Filter*

According to Amdahl's law, the speedup is limited by the serial portion in the algorithm. By analyzing the LETKF algorithm, we understand that this algorithm has a large number of arithmetic operations that can be executed simultaneously, which promises a high level of data parallelism. Table I lists the two major parts of the LETKF algorithm, the global analysis and the local analysis. It is worth noting that the two analysis parts are comprised of basic matrix operations. Within the local analysis each grid point is updated with operations on matrices and vectors formed from its neighbor points. This is very similar to video game image rendering, which computes each image pixel with information associated with surrounding points. At each time step, all grid points can be updated simultaneously and independently, with small matrices and vectors attached to them. In a single core CPU, all grid points are sequentially updated. However, the many-core GPU is able to accomplish this task in parallel, as illustrated by Figure 1.

Table I. Global and local analysis of LETKF, and its operation types

	Operation Type	Formula	Size
golbal analysis	average matrix col-wise	$\bar{\mathbf{x}}^b \leftarrow \{\mathbf{X}^{b(i)}\}$	$n^2 \times 1 \leftarrow n^2 \times k$
	matrix minus vector col-wise	$\mathbf{X}^b \leftarrow \{\mathbf{X}^{b(i)} - \bar{\mathbf{x}}^b\}$	$n^2 \times k \leftarrow n^2 \times k$
	linear mapping	$\bar{\mathbf{Y}}^{b(i)} \leftarrow \{\mathcal{H}(\mathbf{X}^{b(i)})\}$	$m^2 \times k \leftarrow n^2 \times k$
	average matrix col-wise	$\bar{\mathbf{y}}^b \leftarrow \{\mathbf{Y}^{b(i)}\}$	$m^2 \times 1 \leftarrow m^2 \times k$
	matrix minus vector col-wise	$\bar{\mathbf{Y}}^b \leftarrow \{\mathbf{Y}^{b(i)} - \bar{\mathbf{y}}^b\}$	$m^2 \times k \leftarrow m^2 \times k$
local analysis	matrix product / inverse	$\mathbf{C} \leftarrow \{(\mathbf{Y}^b)^T R^{-1}\}$	$k \times \ell \leftarrow k \times \ell \times \ell \times \ell$
	matrix product / inverse	$\tilde{\mathbf{P}}^a \leftarrow [(k-1)\mathbf{I}/\rho + \mathbf{C}\mathbf{Y}^b]^{-1}$	$k \times k \leftarrow k \times k$
	square root of matrix	$\mathbf{W}^a \leftarrow [(k-1)\tilde{\mathbf{P}}^a]^{1/2}$	$k \times k \leftarrow k \times k$
	mat-mat / mat-vec product	$\bar{\mathbf{w}}^a \leftarrow \tilde{\mathbf{P}}^a \mathbf{C}(\mathbf{y}^o - \bar{\mathbf{y}}^b)$	$k \times 1 \leftarrow k \times k \times k \times 1$
	matrix plus vector col-wise	$\{\mathbf{w}^{a(i)}\} \leftarrow \{\mathbf{W}^{a(i)} + \bar{\mathbf{w}}^a\}$	$k \times k \leftarrow k \times k$
	mat-vec product, sum	$\mathbf{x}^{a(i)} \leftarrow \mathbf{X}^b \mathbf{w}^{a(i)} + \bar{\mathbf{x}}^b$	$k \times k \leftarrow k \times k$

In the global analysis, the vectors of background states $\mathbf{x}^{b(i)}_{[g]}$ are used to calculate its mean $\bar{\mathbf{x}}_{[g]}$ and covariance matrix $\mathbf{X}^b_{[g]}$. Next, the projection operator is applied to $\mathbf{x}^{b(i)}_{[g]}$, which generates the ensemble of observation $\mathbf{y}^{b(i)}_{[g]}$. Also, its mean $\bar{\mathbf{y}}_{[g]}$ and covariance matrix $\mathbf{Y}^b_{[g]}$ are calculated. All the calculations above are done component wise. Therefore, when implementing in both the MPI

DOI: 10.1002/cpe

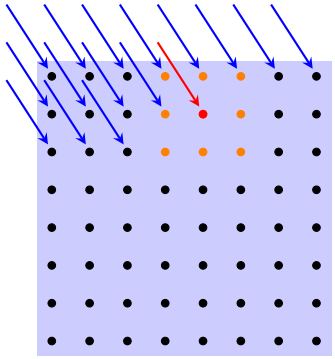

Figure 1. Multiple threads update each point.

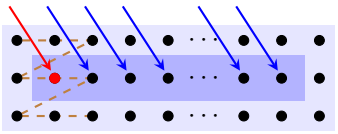

Figure 2. Memory access pattern for a block.

programming model and the CUDA model, there is no data dependency from the point's neighbors. Excellent data parallelism can be achieved in this phase.

On the other hand, the local analysis updates each grid point in the state space, requiring its corresponding m points from its neighbor. As shown in Figure 2, when one thread updates a grid point state from time t to $t+1$, it only requires its neighbors' states at time t. In this schema, each thread can update a grid point without conflicting to other threads. To prepare the local analysis, related rows and columns in $\bar{\mathbf{x}}_{[g]}$ and $\mathbf{X}_{[g]}$ are collected to form local $\bar{\mathbf{x}}$ and $\mathbf{X}$. The same procedure is also applied to the global $\bar{\mathbf{y}}_{[g]}$ and $\mathbf{Y}^b_{[g]}$ to form local $\bar{\mathbf{y}}$ and $\mathbf{Y}^b$. Moreover, for the global observation $\mathbf{y}^{\mathbf{o}}_{[\mathbf{g}]}$, which is of size $\ell_{[g]} \times \ell_{[g]}$. Generally, the $\ell_{[g]}$ is far less than the length of domain size n. We also assume that the observational covariance matrix $\mathbf{R}_g$ of size $\ell^2_{[g]} \times \ell^2_{[g]}$ is known and fixed. Using this information, corresponding local $\mathbf{y}^o$ and $\mathbf{R}$ are formed.

After the local analysis, we obtain the value $\mathbf{X}^a_{[g]}$ for each grid point amongst all members in the ensemble at time t. We can use the shallow water model to forward these analysis state vectors at time t to the next time step $t+1$, and get the new background state vector $\mathbf{X}^b_{[g]}$. This complete one cycle of model simulation with data assimilation.

2.2. *GPU Architectures*

Better understanding of the GPU architecture enables efficient conversation between the software and the hardware. The first language that helps maximize performance is the communication with the memory pipeline. In graphics memory there are five levels of memory, each of which exist entirely on the graphics card. First is the large global memory, which can reach up to 6GB with current graphics cards. Accessing this memory is expensive and should be avoided when possible; however, with the Fermi architecture, two new levels of cache can be used to optimize problems that only fit into global memory. Global memory is accessible from every thread in a kernel. Next is the fast shared memory. This memory can be the key to maximizing performance on the GPU. The downside is the limited amount of shared memory available, which is 16KB or 48KB per block. This memory is only accessible from threads within a block. The third level is local memory. This memory resides on global memory, but can be as fast as shared memory with the usage of registers. Local memory is local to each thread. The fourth level of memory is registers. Registers are a limited resource accessed by each thread. Reducing the number of registers used by each thread can increase the data parallelism of a kernel. The final level of memory is the cache, which only exists in the Fermi architecture. Fermi's cache is out of the scope of this study.

The second language that is significant to the conversation is the GPU scheduler. The main compute units in the graphics architecture are the streaming multiprocessors. When submitting kernels to the GPU, each kernel generates a user-defined number of blocks, which are scheduled onto each streaming multiprocessor. Next, each multiprocessor uses its multiple cores to process the user-defined number of threads per block. These threads are then split up into warps, which are scheduled and processed in parallel by the cores of the streaming multiprocessor. Each warp

DOI: 10.1002/cpe

contains 32 threads. Finding the correct combination of threads per block t_b and number of blocks can optimize the performance of an application and optimize the memory throughput of the kernel. To optimize the throughput, GPUs have the ability to schedule multiple blocks simultaneously on a single streaming multiprocessor. A block that is scheduled on a streaming multiprocessor is said to be an active block. The number of active blocks that a multiprocessor can handle is defined by the capabilities of the device, for instance a Fermi or GTX 200 series card can have up to 8 active blocks for a single multiprocessor. The number of active blocks that can be run on a multiprocessor is bound to the resources used by each block. Therefore, the number of registers per thread r_t, number of threads per block t_b, and the amount of shared memory per block s_b determines the number of active blocks. To help determine the number of active blocks, there are a series of equations that are helpful. These equations can determine the most effective domain size to maximize the memory throughput.

Finding the proper number of threads per block and total number of blocks helps determine the correct decomposition for a GPU kernel. Fully occupying the GPU helps to determine these values and enables maximum utilization of the graphics hardware. Occupancy O is the ratio of the number of active warps per multiprocessor n_w and the number of warps allowed per multiprocessor w_p. n_w is found by multiplying the number of active blocks per multiprocessor n_b by the number of warps per block w_b. The number of active blocks per multiprocessor is found based on the limitations of the hardware. The limitations are expressed in the following equation: $\min(\frac{w_p}{w_b}, \frac{r_p}{r_b}, \frac{s_p}{s_b}, b_p)$, where r_p is the number of registers per multiprocessor, r_b is the number of registers per block, s_p is the amount of shared memory per multiprocessor, s_b is the amount of shared memory per block, and b_p is the maximum number of active blocks per multiprocessor. Example values of these variables can be found in Table II. NVIDIA has provided an excel spreadsheet to help determine the effect various decompositions have with the graphics card [2].

Table II. Notations for number of active blocks calculation

Factors	Notation	Example Values	
		Fermi	GTX 200 Series
Maximum Active blocks per multiprocessor	b_p	8	8
Threads per warp	t_w	32	32
Warps per multiprocessor	w_p	48	24
Registers per multiprocessor	r_p	32768	16384
Shared Memory (bytes) per multiprocessor	s_p	49152	16384
Warps per block	w_b	*user defined*	*user defined*
Registers per block	r_b	*user defined*	*user defined*
Shared memory per block	s_b	*user defined*	*user defined*
Threads per block	t_b	*user defined*	*user defined*
Registers per thread	r_t	*user defined*	*user defined*
Number of Active Blocks per multiprocessor	n_b	*Equation 5*	*Equation 5*
Number of warps per block	w_b	$\frac{t_b}{t_w}$	$\frac{t_b}{t_w}$

$$n_b = \min(\frac{w_p}{w_b}, \frac{r_p}{r_b}, \frac{s_p}{s_b}, b_p) \quad (5)$$

$$n_w = n_b \times w_b \quad (6)$$

$$O = n_w / w_p \quad (7)$$

DOI: 10.1002/cpe

3. OPTIMIZATION TECHNIQUES

3.1. Decomposition

Graphics memory and system memory are located in two separate places, and at many times the graphics memory is much smaller (1GB) than the large system memory (8GB). Therefore large problems that can fit into system memory, may not necessary fit onto the graphics memory. LETKF tends to have very large domain sizes, for example, a typical domain size can take up to 8 GB of memory (or more). To solve this problem, we decompose the domain so that MPI distributes data that fits nicely on the graphics memory.

Our experiments thus far have decomposed these problems into small chunks of size $N_x \times N_y$ for example, $N_x = 256$ and $N_y = 256$. Each element looks at ℓ neighbors within some radius, in our experiment the radius is set to one, which gives us nine total neighbors per element. There are N_{ens} ensembles of these chunks, which in our experiment is set to thirty-two. We have determined that these chunks fit within our graphics global memory, which has up to 868 MB. We then decompose these smaller chunks into thread-level blocks. In order to compute matrix C, we take two matrices, Y^b and R and multiply them. In the global GPU domain, $Y^b = N_{ens} \times N_y \times N_x$, $C = N_{ens} \times \ell \times N_y \times N_x$, $R = \ell \times \ell$. Every thread is responsible for the local matrix in Y^b, consisting of ℓ elements. This local matrix is multiplied to R, resulting in ℓ points for C, see Figure 3. In order to represent this structure in CUDA, we add three dimensions for decomposition. The first level is responsible for the number of threads per block, which is N_x. The second level is responsible for N_y blocks, which forms a single dimension for the total number of blocks. The third level is the N_{ens}, which represents the second dimension of a block. Therefore, in our case we are forming a two dimensional number of blocks, and a one dimensional number of threads per block. The total number of threads spawned on the GPU is equal to $N_x \times N_y \times N_{ens}$, which is equal to $256 \times 256 \times 32 = 2,097,152$ threads. By doing this kind of decomposition every thread is only responsible for a single element in matrix C, which simplifies our graphics kernels, which helps reduce the number of resources used per block. All strategies used above apply to calculating matrix P.

3.2. Memory Access Pattern

In the graphics architecture it is very important to reduce memory transactions by coalescing memory access. If memory is not coalesced, then extra memory transactions are done, causing a significant reduction in the performance. In this section we focus on the pattern used in the first dimension, which is the threads within a block.

To calculate C, matrices R and the transpose of Y^b are stored into global memory and are shared amongst all threads in the kernel. From our decomposition, every thread in a block has a single element such that each successive thread accesses the successive element of the previous thread, see Figure 2. By following this access pattern, we avoid threads from accessing the same element, while at the same time aligning memory accesses [1]. Figure 3 illustrates the update for each element in matrix C. Once C is calculated, special consideration must be done to ensure efficient storage, such that memory is coalesced, and data is prepared for the next phase. In order to satisfy these conditions we analyze the next phase in the assimilation algorithm. Matrix P is calculated by multiplying C and the transpose of Y^b. After calculating each element of C, we store the matrix such that every thread has close proximity to its ℓ neighboring points. By storing C in this fashion, we ensure that threads accessing elements in parallel do not conflict and are within the same block of memory, see Figure 4.

To calculate P, we multiply matrices Y^b and C. This calculation has similar attributes to that of calculating matrix C; therefore, we use a similar decomposition technique as seen above. Figure 4 describes the calculations done for P. Ultimately, each thread in C calculates N_{ens} points in P. A single point in P is calculated by multiplying submatrix C by its corresponding local matrix in Y^b, shown in Figure 4. As before, we store matrix P to optimize for the next step in the algorithm. More study needs to be done regarding the storage of matrix P.

DOI: 10.1002/cpe

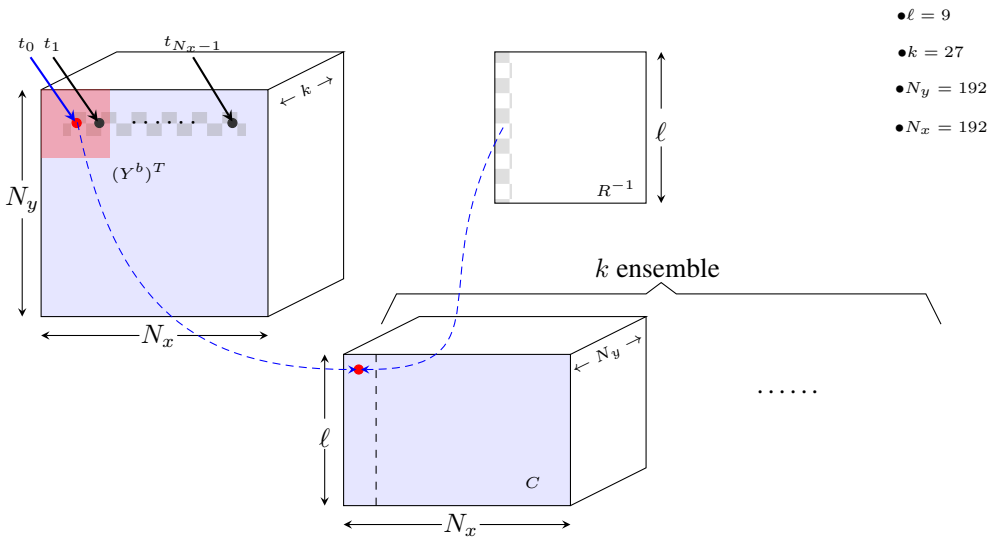

Figure 3. Memory access pattern for a block.

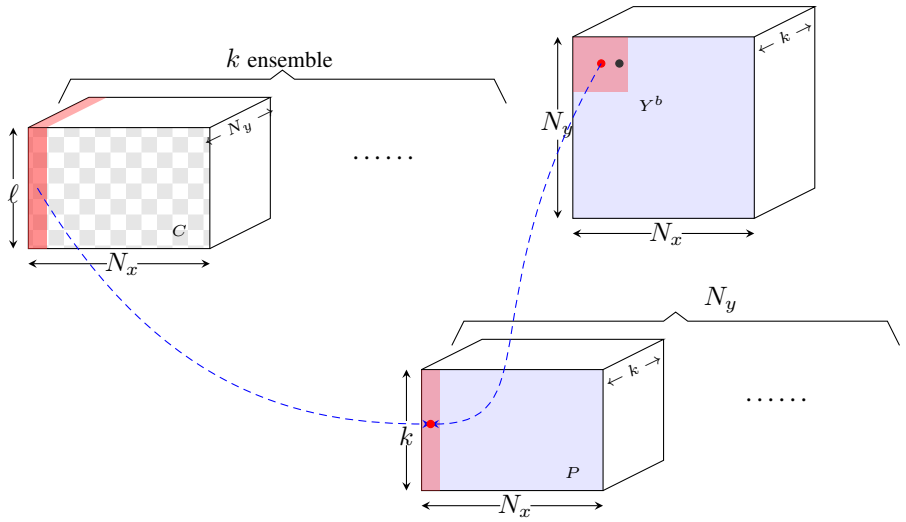

Figure 4. Memory access pattern for a block.

3.3. Shared Memory

Shared memory is the most significant performance improvement available for the GPU. It is similar to optimizing for CPU L1 cache. The main problem with GPU shared memory is its size. On the Fermi architecture, GPUs only have 48KB, and on the GTX 200 series only 16KB. So in order to maximize the performance, blocks must be decomposed such that their memory can be placed into the fast shared memory. Shared memory is local to a single block of threads on the GPU.

In the previous section, decomposition was discussed and enables us to make use of fast shared memory. To calculate matrix C, we store the entire matrix R and sub domain of Y^b for a single block into shared memory, which is represented by the region shadowed with checkerboard pattern in Figure 3. We then do the matrix multiplication on these matrices in shared memory and store the final values into matrix C in global memory.

For calculating a single block of elements of matrix P, we store one ensemble of matrix Y^b for a block from global memory to shared memory. In order to calculate the entire matrix P we iterate over the ensembles loading each ensemble into shared memory for each block. A single iteration

DOI: 10.1002/cpe

of this process is represented from the region shadowed with checkerboard patten in Figure 4. One aspect that is important to note when accessing shared memory is to avoid accessing words in the same bank of memory. If threads within a half warp are accessed in this fashion, then a bank conflict occurs, which increases the number of memory transactions. Because of the way we stored matrix C from our decomposition, we prevent causing bank conflicts.

3.4. Memory Throughput

Memory throughput is maximized based on the number of active blocks because as blocks are being executed on a multiprocessor, other blocks can fetch memory as needed, which hides the slow memory transactions. Equations 5 to 7 help us determine the number of active blocks a kernel is using [2]. The main factors to these equations are t_b, s_b, and r_t. These values can be tweaked to optimize the number of active blocks. See Figure 5.

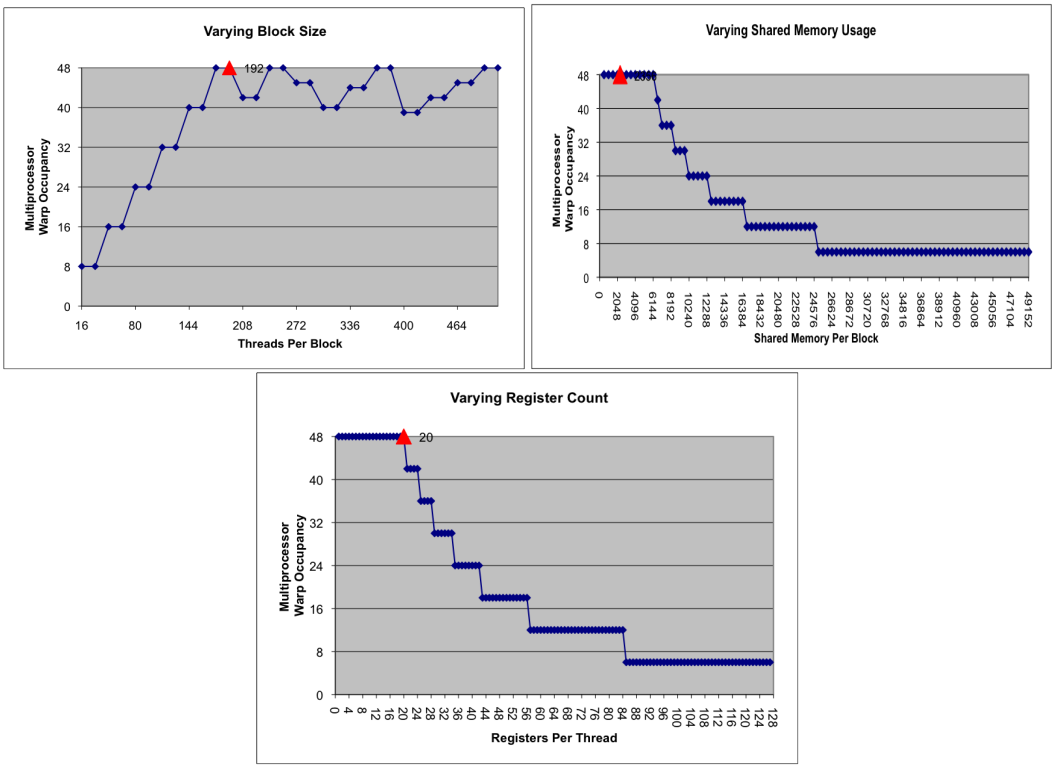

Figure 5. Occupancy calculations for Fermi using t_b, s_b, r_t [2]

In order to maximize the memory throughput of our kernels, we want to maximize the number of active blocks being executed. Using the above calculations and a domain size of 256×256, calculating matrix P uses 6 out of the possible 8 active blocks on the Fermi and 3 out of 8 on the GTX 200 series. This reduction in active blocks is mainly caused by the limited number of registers on each multiprocessor. Each thread is using 20 registers, which equates to 5120 registers. On the GTX 200 series, there are a total of 16384 registers per multiprocessor, which indicates that at most there can be 3 active blocks. So in order to ensure more blocks are active, we must either reduce the domain size or reduce the number of registers. We decided to reduce the domain size by a factor of 32. It is important that the value be divisible by 32 to maximize active blocks based on warps. From this knowledge our next closest domain size is 192. Analysis shows a domain size of 192×192 has an increase in the number of active blocks from 3 to 4 for GTX 200 series cards and an increase from 6 to 8 on Fermi, which maximizes the number of active blocks on the architecture. The cause of the dramatic difference in active blocks between the two architectures is the large increase in the resources available for the Fermi architecture. Fermi has twice as many registers, three times as

DOI: 10.1002/cpe

much shared memory, and a higher volume of warps per multiprocessor. These factors enable our algorithm to have a good fit at the lower domain size. In order to increase the GTX 200 series card's active block usage, we would have to decrease the number of registers used.

4. PERFORMANCE STUDY

For our performance study we have two different setups. The first is a q6600 @ 2.4GHz with 4 GB DDR2 memory and a GTX 260 graphics card (GTX 200 series) running CUDA version 2.1. The second is using a q6600 @ 2.4 GHz with 4 GB DDR2 memory and a GTX 460 graphics card (Fermi) running CUDA version 2.1. See Table III for details on the hardware specifications between the two graphics cards. Although the q6600 has four cores, the purpose of this performance study is to analyze the different optimization techniques used for graphics cards. Therefore, our implementation is only utilizing a single core to simplify comparison and to use the CPU time as a constant factor between the GPU implementations and changes in domain size.

Table III. Graphics hardware comparison

Graphics card	EVGA GTX 260	EVGA GTX 460
Architecture	GTX 200 series	Fermi
Cuda cores	216	336
Streaming Multiprocessors	27	7
Cores per Multiprocessor	8	48
Core clock speed	1242 MHz	1526 MHz
Global memory size	868 MB	1024 MB
Effective Memory Clock	1998MHz	3800MHz
Shared memory per multiprocessor	16 KB	48 KB
Registers per multiprocessor	16384	32768
Warps per multiprocessor	16384	49152

Table IV. Speedup analysis

Kernel	Calculation	GTX260 vs CPU	GTX460 vs CPU	GTX460 vs GTX260
Naive	Matrix C	15.0x	93.7x	6.2x
Shared Memory	Matrix C	54.0x	77.5x	1.4x
Decomposition-256	Matrix C	225.8x	181.1x	0.8x
	Matrix P	52.5x	80.8x	1.5x
	Combine C and P	70.7x	99.6x	1.4x
Decomposition-192	Matrix C	211.4x	175.4x	0.8x
	Matrix P	54.3x	75.1x	1.4x
	Combine C and P	72.1x	92.7x	1.3x

Naive – 256×256 domain, no optimizations, direct CPU to GPU port. This kernel was programmed prior to the release of Fermi. Interesting to see the performance of the GTX 460 on this problem. Without any optimizations and no shared memory, the GTX 460 was able to achieve over $90\times$ performance and $6.2\times$ faster than the GTX 260. The main causes of this significant speed-up is due to the large amount of resources, higher clock speeds, and the creation of a cache hierarchy provided by the Fermi architecture.

DOI: 10.1002/cpe

Shared Memory – 256×256 domain, added shared memory for matrices Y^b and R. Another interesting factor is the decrease in performance from the naive implementation for the Fermi card. This may be directly caused by the new cache hierarchy that Fermi has integrated into their architecture. The cause of the slow speeds for this kernel is caused by the extra memory transactions done when loading values into shared memory. Each thread must load k ensembles from global memory into shared memory. The number of calculations done in shared memory is not significant enough to offset the number of transactions completed.

Decomposition-256 – 256×256 domain, shared memory for matrices Y^b and R for matrix C calculation, and shared memory for matrix Y^b. Added level of decomposition by increasing the dimensionality of the number of blocks by the number of ensembles in the problem. This effectively reduced the number of resources used, but drastically increased the number of threads generated. The large increase in performance by this kernel is directly related to the decrease in the number of memory transactions done per thread. The transactions are decreased because of the reduction in the number of registers and the elimination of the loop over the k ensembles. In the naive and shared memory kernels, each thread used 25 and 22 registers respectively, which decreased the number of active blocks. With the added level of decomposition, this value was reduced to only 8 registers per thread, increasing the memory throughput of the kernel. On the down-side it spawned more blocks for execution, which the GTX 200 series handled nicely, but was not handled as nicely on the Fermi because of the reduced number of multiprocessors.

Decomposition-192 – 192×192 domain, same as *Decomposition-256*, except with a reduced domain size from 256×256 to 192×192. The purpose of this design is to analyze the affect of increasing the number of active blocks per multiprocessor. This decrease in threads enables a greater number of active blocks, which should increase the memory throughput. The problem size between the two kernels is decreased by a factor of 1.7. The GPU time between the two kernels indicate a faster compute time of $1.7\times$. Therefore, although the performance is better between the two kernels, the main cause is due to the reduction in problem size. Although *Decomposition-192* maximizes the number of active blocks per multiprocessor, the effect is not significant to increase the performance between the two kernels. This is because the occupancy between the two kernels is the same; therefore, we are already utilizing the GPU to its maximum potential.

5. SUMMARY

From our analysis we can determine that there are significant opportunities for the Local Ensemble Transform Kalman Filter on GPUs. By taking advantage of the locality of the algorithm, we can spawn a massive number of threads on the GPU, and do the local analysis simultaneously for each grid point on the problem domain. This promises that GPUs accelerate the LETKF algorithm, and can be used on problems that exhibit high time efficiency demands.

In our optimization techniques, we found that by increasing the decomposition we reduced resources used per thread and simplified the code. On the GTX260 we saw a speedup of $4\times$ between the *Shared Memory* and *Decomposition-256* kernels, and a speedup of $15\times$ between the *Naive* and *Decomposition-256*. See Table IV. The decomposition of the problem significantly impacts the performance that is seen from the graphics card.

From the performance study, we can conclude that higher speed up can be achieved by using shared memory and simplifying the kernel with decomposition. One interesting aspect is it is not always necessary to increase the number of active blocks for a kernel, as seen in comparing *Decomposition-192* with *Decomposition-256*. On a particular graphics architecture, a program reaches its highest potential when the graphics card's occupancy is full. Another aspect found from our study is a dramatic increase in performance for the *Naive* kernel between the two architectures. The two architectures have a similar number of cores, so the cause of this performance difference is due to the introduction of cache in the Fermi architecture. More study is needed to be done on this aspect.

Regarding future work, one of the most interesting aspects of the results found is the difference in the architectures. Fermi needs further investigation on strategies to optimize code. More specifically,

DOI: 10.1002/cpe

Fermi's increase in resources enables a greater number of threads per block, and encourages it, particularly with the increase of the number of cores per multiprocessor (from 8 to 48). Also the cache and dual warp scheduler on the Fermi needs to be properly analyzed. This study does not exhaust the optimizations that can be done to further increase performance.

In the current study, the CPU code utilizes only a single thread. It would be an interesting problem to fairly compare the LETKF algorithm between the CPU and GPU by implementing a CPU multi-threaded version of the code.

In this study the main contribution is to analyze the strategies for utilizing data parallelism on the GPU. We focus on calculating matrices C and P because the matrix operations of the local analysis are similar to image rendering. These matrices feature ensembles, which increase the computational complexity and data requirements. By calculating and analyzing the matrices on the GPU we demonstrated optimal strategies to enhance data throughput on the GPU. The final portions of the algorithm may also benefit from the GPU, so in the future we intend on finishing the remainder of the local analysis and compare the results with the CPU implementation.

Acknowledgment: The authors would like to thank Dr. Kostas Kalpakis for his guidance with the data assimilation algorithm.

REFERENCES

1. Nvdia Corp. online document 2010. *NVIDIA CUDA C Programming Best Practices Guide version 2.3*. http://developer.nvidia.com.
2. Nvdia Corp. online document. 2010. *NVIDIA CUDA Occupancy Calculator*. http://developer.download.nvidia.com/compute/cuda/3_2_prod/sdk/docs.
3. TOP500 Supercomputer Site. 2011. *TOP500 Supercomputer Novermeber 2010 List*. http://www.top500.org/lists/2010/11.
4. Brain R. Hunt and et. al. 2007. *Efficient data assimilation for spatiotemporal chaos: A local ensemble transform Kalman filter*. Phy. D, pp112-126.
5. Takemasa Miyoshi and Shozo Yamane. 2007. *Local Ensemble Transform Kalman Filtering with an AGCM at a T159/L48 Resolution*. American Meteorological Society, Vol. 135, pp3841-3861.
6. Istvan Szunyogh and Eric J. Kostelich and G. Gyarmati and et.al. 2005. *Assessing a Local Ensemble Kalman Filter: Perfect Model Experiments with the National Centers for Environmental Prediction Global Model*. Tellus, Vol. 57A, pp528-545.
7. Cleve Moler. 2000. *Experiments with Matlab, Chap 16, Shallow Water Equations*. http://www.mathworks.com.
8. Victor Lee and Changkyu Kim and Jatin Chhugani and Michael Deisher and et. al. 2010. *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU*. ACM SIGARCH Computer Architecture News, Vol. 38, pp451-460.
9. Geir Evensen. 2007. *Data Assimilation: The ensemble Kalman filter*. Springer.
10. Eugenia Kalnay. 2003. *Atmospheric Modeling, Data Assimilation and Predictability*. Cambridge University Press.
11. Ming Xue and Kelvin K. Droegemeier and Daniel Weber. 2007. *Petascale Computing: Algorithms and Applications,*. chapter Numerical Prediction of High-Impact Local Weather: A Driver for Petascale Computing. Chapman & Hall/CRC.

DOI: 10.1002/cpe