# Unstructured Grid Applications on GPU: Performance Analysis and Improvement

Lizandro Solano-Quinde
Electrical and Computer Engineering, Ames Laboratory
Iowa State University
Ames, IA 50010
lsolano@iastate.edu

Zhi Jian Wang
Department of Aerospace Engineering
Iowa State University
Ames, IA 50010
zjw@iastate.edu

Brett Bode
National Center for Supercomputing Applications
University of Illinois
Urbana, IL 61801
bbode@ncsa.illinois.edu

Arun K. Somani
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA 50010
arun@iastate.edu

## ABSTRACT

Performance of applications running on GPUs is mainly affected by hardware occupancy and global memory latency. Scientific applications that rely on analysis using unstructured grids could benefit from the high performance capabilities provided by GPUs, however, its memory access pattern and algorithm limit the potential benefits.

In this paper we analyze the algorithm for unstructured grid analysis on the basis of hardware occupancy and memory access efficiency. In general, the algorithm can be divided into three stages: cell-oriented analysis, edge-oriented analysis and information update, which present different memory access patterns. Based on the analysis we modify the algorithm to make it suitable for GPUs. The proposed algorithm aims for high hardware occupancy and efficient global memory access. Finally, through implementation we show that our design achieves up to 88 times speedup compared to the sequential CPU version.

## Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*; J.2 [**Physical Sciences and Engineering**]: Aerospace

## General Terms

Performance, Unstructured Grid, CUDA, GPGPU, GPU

## 1. INTRODUCTION

Graphics accelerators, developed for programmable video capabilities, turned into manycore processors, known as Graphics Processing Units (GPUs). Current GPUs integrate hundreds of processing cores and achieve up to 1 TFlop for single precision and over 80 GFlops for double precision. In addition to this tremendous computational power, GPUs became fully programmable, which makes GPUs amenable for accelerating massively-parallel general purpose applications, opening a new application field to GPUs called *General-Purpose Computation on GPUs* (GPGPU) [1].

Also, the availability of GPGPU programming languages have consolidated the adoption of GPU technology as an alternative for accelerating general-purpose applications. GPGPU programming languages simplify the implementation of general-purpose applications on GPUs. Computing Unified Design Architecture (CUDA) is a proprietary GPGPU-oriented programming language developed by NVidia that adds extensions to the C programming language to ease the coding of general-purpose programs on NVidia GPUs [2].

There are reports of successful implementation of applications running on GPU-accelerated systems [1, 3]. These applications range on a variety of fields, such as: signal and image processing, database acceleration, financial analysis, chemistry, oil industry, and, in general, in the High Performance Computing (HPC) Industry.

Computational Fluid Dynamics (CFD) is a field suitable for GPGPU, where fluid flow analysis is performed on a surface represented as a grid using numerical methods. In general, unstructured grids are used due to their flexibility and accuracy; however, unstructured grid-based analysis presents more complex internal structures and algorithms, which imposes challenges for GPGPU computing.

This paper presents the analysis and implementation of unstructured grid applications on GPUs. Our specific contributions are: (i) present the memory access pattern for grid applications, (ii) propose an algorithm that achieves high hardware occupancy and efficient memory access, (iii) analyze the performance of the proposed algorithm, and (iv) implement the algorithm.



*GPGPU-4*, Mar 05-05 2011, Newport Beach, CA, USA

The rest of the paper is organized as follows. Section 2 presents the related work, Sections 3 and 4 present the grid-based analysis algorithm and its memory access pattern. Next, Sections 5 and 6 present the performance analysis of the GPU implementation and introduce modifications to the algorithm aiming for improving its performance, in Section 7 the results for the implementation of the proposed algorithm are presented and Section 8 concludes with the conclusions.

## 2. RELATED WORK

Unstructured grids are very important in parallel computing for several fields where the computational intensity can be very uneven, resulting in poor load balancing for regular grids. A good example is weather modeling where it is needed a much finer mesh in active areas (where there are storms) than where the atmosphere is relatively stable. CFD is another example where fluid flow analysis is done using unstructured grids.

Unstructured grid based analysis methods on shared memory and distributed memory systems have been largely studied in the last decade [4, 5]. However, shared and distributed memory systems are fundamentally different from GPUs. A GPU is a SIMT (Single Instruction Multiple Thread) engine, whereas shared and distributed memory systems are MPMD (Multiple Program Multiple Data) engines. However, the common aspect of these parallel engines is that in both of them the grid application is limited by memory latency [6, 7].

There are several successful efforts by scientific community to implement unstructured grid applications on GPUs [8, 9], however, none of these analyze performance or provide an algorithm suitable for GPUs. In this paper, the performance of the implementation is analyzed and an algorithm suitable for GPUs is proposed.

## 3. UNSTRUCTURED GRID APPLICATIONS

Many fields of scientific research rely on simulations that require the analysis of surfaces. In order to be tractable by computational methods, i.e. to numerically solve partial differential equations, surfaces are discretized into small cells that form a mesh or grid. Depending on the complexity of the problem to represent and solve, a structured or unstructured grid is utilized.

In a structured grid all its interior cell vertices belong to the same number of cells, whereas in a unstructured grid every cell vertex is allowed to belong to different number of cells, see Figure 1.

Algorithms are more efficiently implemented in structured grids, and data structures to handle the grid are easy to implement; however, structured grids present poor accuracy if the problem to be solved has curved internal or external boundaries. On the other hand, unstructured grids present more flexibility and higher accuracy to represent problems that have curved boundaries; however, the data structures to handle it are not easy to implement, and also explicit neighboring information should be stored [7]. In general unstructured grids are more utilized because of their flexibility and higher accuracy.

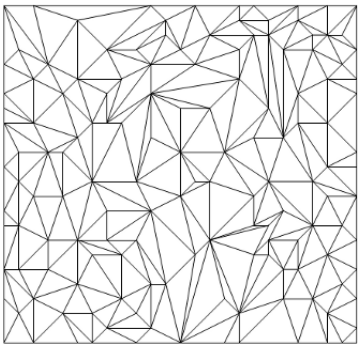


Figure 1: Unstructured grid representation of a surface

This document focuses on unstructured grids, therefore, the terms unstructured grid and grid are used interchangeably in the next sections.

Every cell in a grid is defined by its shape, i.e. number of sides or faces, the number of solution points in every face, i.e. $k$, and the number of solution points inside the cell. The total number of solution points per cell is given by:

$$\#SolutionPoints = faces \times (k-1) + InnerPoints \quad (1)$$

Figure 2 shows a triangular cell with four solution points per face ($k = 4$) and one inner solution point.

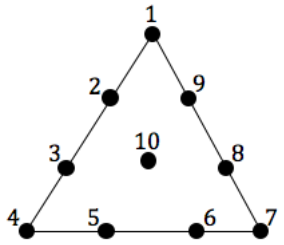



Figure 2: A triangular cell with $k = 4$

Each solution point has a set of values that represent the local magnitudes for the set of variables to analyze or parameters at that point, i.e. $v_{i,j}$ with $1 \leq i \leq \#SolutionPoints$ and $1 \leq j \leq \#Parameters$. Depending on the research field the parameters may represent pressure, viscosity, velocity, etc.

Analysis using an unstructured grid is implemented as an iterative method where the values of the variables at each solution point are updated until converges to the solution or reaches a number of iterations. The operations that are carried out in every iteration can be divided into three parts:

- *Local cell analysis*: obtains a coefficient for each solution point based only on the interaction with the other solution points in the same cell.
- *Neighbor cell analysis*: computes a coefficient for each solution point based on the interaction with its neighbor solution point.

- *Update local magnitudes*: the local value of the magnitude at the solution point is updated using the two previously computed coefficients.

Next, Algorithm 1 presents the main algorithm for analysis based on unstructured grids.

**Algorithm 1** Analysis using an unstructured grid

```
1: counter ← 0
2: repeat
3:    {Cell-oriented Analysis}
4:    for all cells in grid do
5:       for all solutionPoints in currentCell do
6:          Compute local coefficient based on information
            on solution points within the same cell
7:       end for
8:    end for
9:    {Neighbor or Edge-oriented Analysis}
10:   for all edges in grid do
11:      for all solutionPoints in currentEdge do
12:         Compute local coefficient based on information
            on the neighbor solution point
13:      end for
14:   end for
15:   {Updates variables at solution points and checks for
      convergence}
16:   for all solutionPoints in grid do
17:      Update local magnitudes utilizing local coefficient
         computed in the previous steps
18:      Check for convergence
19:   end for
20:   counter ← counter + 1
21: until ( Converges ) or ( counter = Max )
```

As it can be seen in Algorithm 1, the three main stages perform computations based on information stored in main memory, such as the solution point variables, geometry information, and a set of parameters for cell-oriented or neighbor-oriented (edge-oriented) analysis. What is interesting to notice is that although solution point variables and parameters are heavily used in all three main stages, they are accessed with different patterns at every stage. These memory patterns limit data locality between and inside the stages, diminishing efficiency of data caches for reducing memory latency. Therefore, performance of the grid analysis algorithm is limited by memory latency.

# 4. MEMORY ACCESS PATTERN

In Section 3, the main algorithm for analysis using unstructured grids was introduced, which because of the data access patterns of its main stages is limited by memory latency. In this section we describe the access pattern for each of the three main stages in Algorithm 1.

In cell-oriented analysis, a set of coefficients for each solution point is computed based on its own information as well as the information of the solution points that belong to the same cell. As shown in Figure 3 accessing the solution point information is performed in two steps: the first step involves retrieving the pointer to the beginning of the cell in the array of solution point variables, and the second step involves accessing sequentially all the information in the current cell. Because all the cells store the same amount of information, the array with pointers to the beginning of the cell, i.e the array of cells in Figure 3, it is not needed. Instead the beginning of a cell in the array solution point variables can be computed as:

$$Ptr = \#SolutionPoints \times \#Parameters \times (\#Cell - 1)$$

As it can be noticed, the solution point variables in every cell are read once and utilized several times, i.e $\#SolutionPoints$ times. Clearly on a uni-threaded solution, cache memories are useful to exploit temporal and spatial locality. Unfortunately, depending on the design of a multi-threaded solution cache memories can exploit only spatial locality, because it is likely that information in cache is replaced as required by different threads. Therefore, the expected performance gain by a multi-threaded solution could be drastically reduced.

In edge-oriented analysis, a set of coefficients for each solution point is computed based on its own information and the information of its neighbor solution point, see Figure 4.

Unlike cell-oriented analysis that traverses the grid at cell-level, edge-oriented analysis traverses the grid at edge-level. Figure 5 shows that accessing the solution point information is done in three steps: the first step involves retrieving the pointer to the solution point, in the second step the pointer to the left and right solution point variables, and the third step involves accessing the two solution points variables.

Alike cell-oriented analysis, the first step is trivial as the pointers to the solution points can be easily computed as $2 \times (\#edge - 1)$.

Unlike cell-oriented analysis, left and right solution point variables are not physically adjacent, and information is read and used only once, hence, either on a uni-threaded or multi-threaded solution the cache memories do not help to reduce memory latency.

In the last stage the solution point variables are updated utilizing only current solution point information and coefficients, i.e. read and utilized once. Since coefficients and solution point variables arrays are processed sequentially, cache memories can take advantage of spatial locality, and by this way help to reduce memory latency for both uni-threaded and multi-threaded solutions.

# 5. PERFORMANCE CONSIDERATIONS

In the CUDA platform, for execution purposes thread blocks are assigned to Streaming Multiprocessors (SMs) whereas for scheduling purposes the threads belonging to the same thread block are grouped into *warps*. Instructions of a warp are executed one at a time on all its threads.

GPUs achieve high performance by hiding memory access latency, which is possible by switching warp execution between warps that are waiting for long latency operations to finish and warps that are ready to continue execution. Under this premise performance on GPUs is mainly dependent on SM occupancy and global memory access.

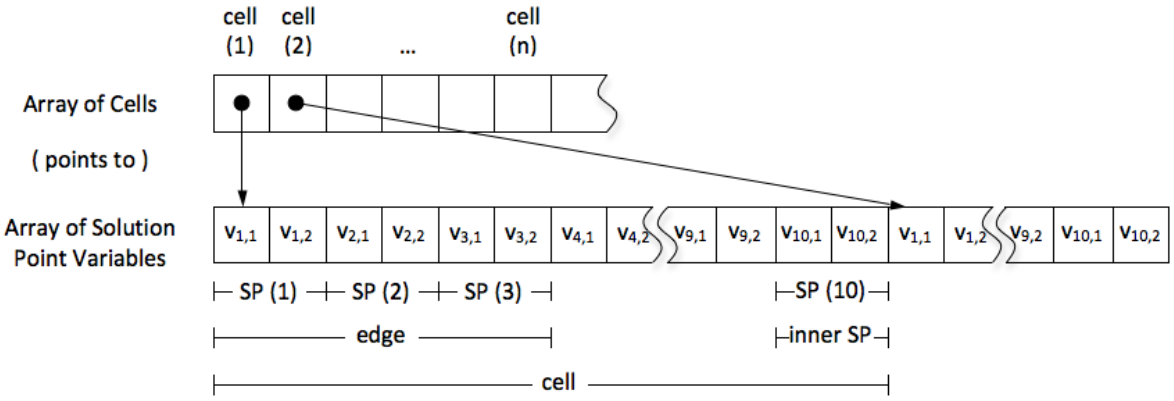



Figure 3: Cell-Oriented Analysis memory access pattern for a grid with 3-sided cells, four solution points per face, one inner solution point and 2 parameters per solution point

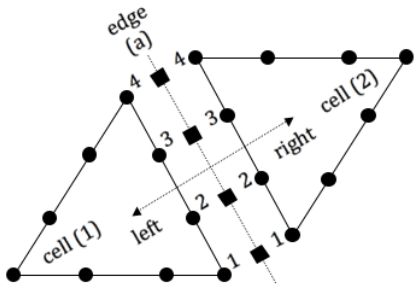



Figure 4: Cell iterations between neighbors

## 5.1 Streaming Multiprocessor occupancy

In a GPU a large number of warps active in a SM, i.e high occupancy, is needed to tolerate long latency operations. The maximum occupancy in terms of number of threads, blocks and warps that can be achieved is determined by hardware specifications [10]. However, achieving maximum occupancy depends on the number of registers and amount of shared memory utilized by each thread block.

The number of registers utilized by the threads in the active thread blocks cannot be greater than the maximum number of registers of the SM. In the same way, the amount of shared memory utilized by the active thread blocks cannot be greater than the total amount of shared memory of the SM. Hence, the number of active thread blocks can be computed as follows.

$$ThreadBlocks = \min\left(\left\lfloor \frac{TotalSharedMemory}{SharedMemoryPerBlock} \right\rfloor, \left\lfloor \frac{TotalRegisters}{RegistersPerBlock} \right\rfloor\right) \quad (2)$$

Now, taking into consideration the maximum number of active thread blocks imposed by hardware specifications, the number of active threads is given by:

$$ActiveWarps = \min\Big(maxActiveWarps, maxTBlocksPerSM \times WarpB, ThreadBlocks \times WarpB\Big) \quad (3)$$

In the previous equation $WarpB$ refers to the number of *warps per thread block*, which depends on the number of threads per block as shown next.

$$WarpB = \left\lceil \frac{ThreadsPerBlock}{ThreadsPerWarp} \right\rceil \quad (4)$$

Equation 3 defines the number of warps that can be active on a SM and it shows that the highest value is limited by register or shared memory usage, by the maximum number of active thread blocks and active warps specified by the hardware implementation.

Table 1 describes the parameters utilized in Equations 2, 3 and 4 that influence the occupancy. The first five parameters in the table are architecture dependent, whereas the last two parameters are application dependent.

## 5.2 Global memory access

A GPU implements different types of memory for storing data: global memory, constant memory, texture memory, shared memory and registers. This memory structure allows to reduce global memory accesses and collaboration among threads in the same thread block. In terms of latency, global memory access is the slowest whereas registers are the fastest.

Since the GPU execution model requires that the information is first placed in global memory and then accessed by the GPU application, it is necessary to optimize global memory access. Global memory access can be optimized by achieving peak bandwidth and by reducing the number of accesses.

Although GPU provides large bandwidth for global memory operation, the access pattern of the threads of a warp can re-

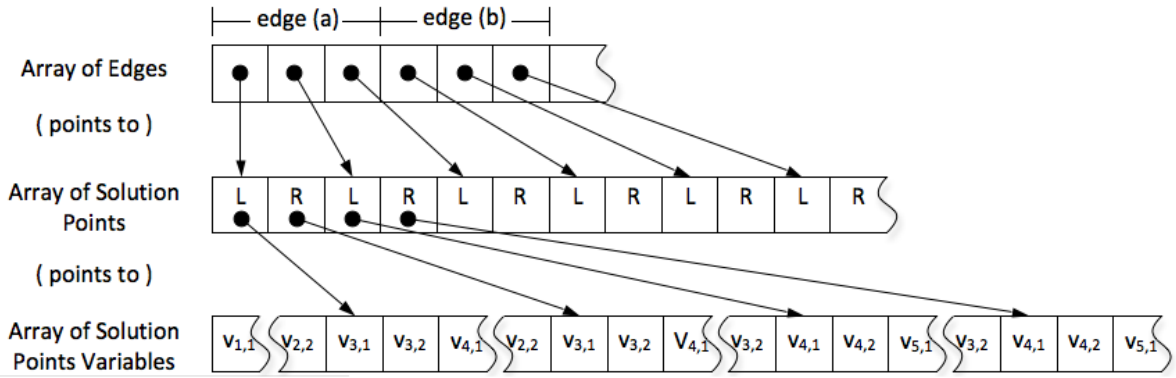



Figure 5: Edge-Oriented Analysis memory access pattern for a grid with 3-sided cells, four solution points per face, one inner solution point and 2 parameters per solution point

Table 1: Parameters that influence SM occupancy

| Parameter | Description |
|---|---|
| TotalSharedMemory | Total amount of shared memory per SM |
| TotalRegisters | Number of registers available per SM |
| ThreadsPerWarp | Number of threads that are grouped into warps |
| maxActiveWarps | Maximum number of warps that can be active in a SM |
| maxTBlocksPerSM | Maximum number of Thread Blocks that can be active in a SM |
| SharedMemoryPerBlock | Amount of shared memory utilized by a thread block |
| RegistersPerBlock | Number of registers utilized by a thread block |

duce considerably the achieved bandwidth. To achieve peak bandwidth usage, the GPU coalesces warp memory operations into two or four memory transactions depending on the size of the words accessed. Therefore, warp memory access should be organized in such a way that threads access adjacent memory locations. Depending on the memory access pattern the number of memory transactions per warp is limited as follows.

$$2 \leq MemTransactions \leq MaxMemTransactions = ThreadsPerWarp \quad (5)$$

When data is reutilized it is possible to reduce the number of global memory accesses by storing the data either in registers or in shared memory. Shared memory is common for all the threads in the thread block, which allows collaboration among them. Since shared memory is organized in banks, to avoid bank conflicts threads should access data in different banks.

# 6. GPU IMPLEMENTATION

This section presents the implementation and performance analysis of Algorithm 1. Since the performance analysis require hardware-dependent parameters, the NVidia Tesla T10 GPU is used on the remainder of this paper. The technical specifications of the Tesla GPU are shown in Table 2.

## 6.1 Streaming Multiprocessor occupancy

Due to space constraints this section presents the implementation and analysis only for the cell-oriented stage, analysis for the edge-oriented stage is similar.

Table 2: NVidia Tesla T10 Technical Specifications

| Parameter | Value |
|---|---|
| Number of Streaming Multiprocessors | 30 |
| Number of Streaming Processors per SM | 8 |
| Number of 32-bit Registers per SM (TotalRegisters) | 16 K |
| Shared Memory per SM (TotalSharedMemory) | 16 KB |
| Warp Size (ThreadsPerWarp) | 32 |
| Active Warps per SM (maxActiveWarps) | 32 |
| Active Thread Blocks per SM (maxTBlocksPerSM) | 8 |

As mentioned in Section 3, in cell-oriented analysis every solution point computes a coefficient based on its own variables as well as the variables of the solution points in the same cell, hence, there is no collaboration between cells.

The straight forward implementation maps one cell to one thread block, where each solution point is represented by one thread. In this implementation the number of thread blocks is equal to the number of cells and the number of threads per thread block is equal to the number of solution points.

Using Equation 2 and the fact that acording to Table 2 the maximum number of active thread blocks per SM is eight:

$$\begin{aligned} 8 &= \left\lfloor \frac{16 \text{ KB}}{SharedMemoryPerBlock} \right\rfloor \\ &\implies SharedMemoryPerBlock = 2 \text{ KB} \\ 8 &= \left\lfloor \frac{16 \text{ K}}{RegistersPerBlock} \right\rfloor \\ &\implies RegistersPerBlock = 2 \text{ K} \end{aligned} \tag{6}$$

Now, assuming triangular cells and using Equation 1 it is possible to approximate:

$$\begin{aligned} SharedMemoryPerThread &= \frac{2 \text{ KB}}{\#SolutionPoints} \\ &\approx \frac{2 \text{ KB}}{3 \times (k-1)} \\ RegistersPerThread &= \frac{2 \text{ K}}{\#SolutionPoints} \\ &\approx \frac{2 \text{ K}}{3 \times (k-1)} \end{aligned} \tag{7}$$

For a large value of solutions points per edge, i.e. $k = 10$, it is possible to have approximately up to 90 32-bit words for calculations between shared memory and registers, which it is enough. Therefore, it is possible to have eight active thread blocks per SM because shared memory and registers do not impose limitations on the number of active thread blocks per SM.

Finally, from Equation 4 the number of warps per thread block:

$$\begin{aligned} \text{Warps per TBlock} &= \frac{\text{\# Solution Points}}{\text{Threads per Warp}} \\ &= \frac{3 \times (10-1) + \text{Inner Points}}{32} \\ &\approx 1 \end{aligned} \tag{8}$$

Therefore, there are eight active warps per SM, which means that only 25% of the occupancy is achieved by this implementation.

Clearly, it is necessary to increase occupancy to fully hide global memory access. To increase occupancy it is necessary to increase the number of threads per thread block, i.e. to increase the number of cells per thread block. In this case the active thread blocks is given by

$$\begin{aligned} \text{Act. TBlocks} &= \left\lfloor \frac{16 \text{ KB}}{\text{S. Mem per Block}} \right\rfloor \\ &= \left\lfloor \frac{16 \text{ KB}}{\text{S. Mem per Cell} \times \text{Cells per TBlock}} \right\rfloor \\ &\leq 8 \end{aligned} \tag{9}$$

In Equation 9 the number of *cells per thread block* is chosen such that the number of *active thread blocks per SM* does not exceed eight, therefore, it is not limited by either shared memory or hardware specifications. Now, the number of active warps per SM can be computed as follows.

$$\begin{aligned} \text{Act.Warps} &= \text{Active TBlock} \times \text{Warps per ThreadBlock} \\ &= \left\lfloor \frac{16 \text{ KB}}{\text{S. Mem per Cell} \times \text{Cells per T. Block}} \right\rfloor \\ &\times \left\lceil \frac{\text{\# Sol. Points} \times \text{Cells per T. Block}}{32} \right\rceil \\ &\leq 32 \end{aligned} \tag{10}$$

From Equation 10 a good approximation for testing purposes can be derived:

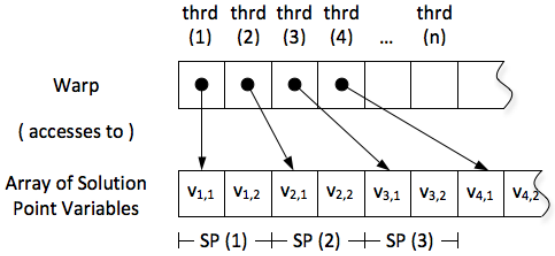
$$\begin{aligned} \text{Act. Warps} &= \frac{512 \times \text{\# Solution Points}}{\text{Shared Mem per Cell}} \\ &= \frac{512}{\text{Shared Mem per Solution Point}} \end{aligned} \tag{11}$$

A similar analysis for the used registers provides the following approximation:

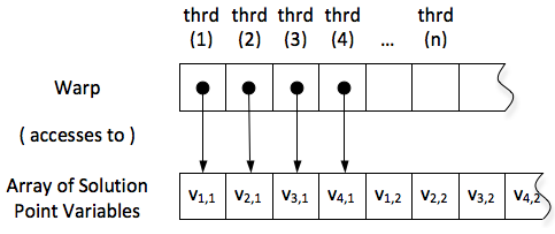
$$\text{Act. Warps} = \frac{500}{\text{Registers per Solution Point}} \tag{12}$$

## 6.2 Global memory access

Section 4 presented the memory access pattern for the cell-oriented analysis. In the GPU implementation every thread accesses the information of a single solution point. According to the algorithm presented in Section 3, every thread accesses non contiguous memory locations, which increases the number of memory transactions non-linearly depending on the number of parameters stored at each solution point. Figure 6a shows non-contiguous memory access for a case with two parameters.




(a) Before data reorganization




(b) After data reorganization

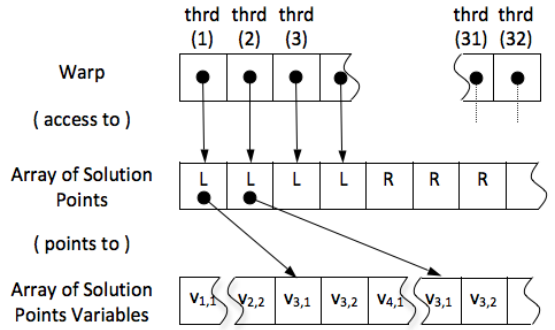
Figure 6: Thread memory access on the cell-oriented stage

In order to access contiguous memory locations, the data have to be reorganized as shown in Figure 6b. The number of memory transactions for the reorganized data implementation can be computed using Equation 13

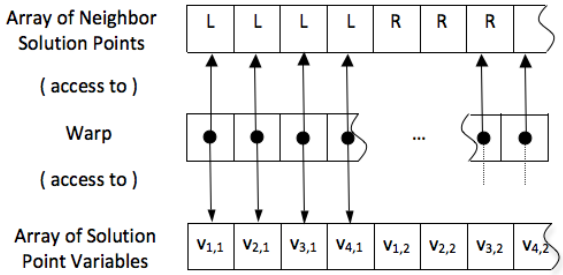
$$\begin{aligned} \#\text{ Transactions} &= 2 \times \left\lceil \#Parameters \right. \\ &\quad \left. \times \frac{1/2 \times \text{Th. per Warp} \times sizeof(float)}{\text{Transaction Size} = 128} \right\rceil \\ &= 2 \times \#Parameters \end{aligned} \tag{13}$$

As it can be seen in Equation 13, for single precision parameters the number of memory transactions for a warp increases linearly with the number of parameters.

Memory access pattern for edge-oriented analysis, is performed in two stages: first the data geometry is read and then the solution point information. As shown in Figure 7a, geometry information is accessed sequentially, therefore, coalescing memory transfers is possible. However, due to the memory access pattern, solution point information is accessed in a random-like fashion, making not possible to coalesce memory transfers, which leads to potentially one memory transaction per thread.




(a) Before data reorganization




(b) After data reorganization

Figure 7: Thread memory access on the edge-oriented stage

Under this considerations, the number of memory transactions is approximated as:

$$\#\text{ Transactions} = 2 \times \#\text{Edges} \times (k-1) \times \#Parameters \tag{14}$$

Intuitively equation 14 represents all the non-coalesced memory accesses. Since the number of memory transactions increase linearly with the total number of threads, the objective is to decrease the number of memory access or in other words avoid the random-like memory access.

The strategy used to reduce the number of memory transactions is depicted in Figure 7b. The main difference is that instead of traversing the grid through the edges, the grid is traversed through cells, therefore, the memory pattern for the solution point variables changes such that the threads read contiguous memory locations. The neighbor pointer for all the solution points is stored in a new structure, and it is accessed sequentially by the threads. However, the neighbor information is still accessed in a semi random-like fashion. In the same way as with the previous scheme the number of memory transactions can be approximated as:

$$\begin{aligned} \#\text{ Transactions} &= \#Parameters \\ &\quad \times \left\lceil \#\text{Faces} \times \#\text{Cells} \right. \\ &\quad \left. + 2 \times \frac{\#\text{Cells}}{\text{Cells per T. Block}} \times WarpB \right\rceil \end{aligned} \tag{15}$$

The terms inside the brackets in Equation 15 represent the neighbor information and the solution point accesses, respectively.

In general the number of memory transactions defined by Equation 14 is bigger that the one defined by Equation 15.

This change in the memory access pattern changes the algorithm described in Section 3, which is described in Algorithm 2.

**Algorithm 2** Unstructured grid analysis on GPU

```
1: counter ← 0
2: repeat
3:    {Cell and Edge Analysis}
4:    for all ThreadBlocks in cudaGrid do
5:       for all cells in currentThreadBlock do
6:          for all solutionPoints in currentCell do
7:             Compute local coefficient based on information
               on solution points within the same cell
8:             Compute local coefficient based on information
               on the neighbor solution point
9:          end for
10:      end for
11:   end for
12:   {Updates variables at solution points and checks for
      convergence}
13:   for all solutionPoints in grid do
14:      Update local magnitudes utilizing local coefficient
         computed in the previous steps
15:      Check for convergence
16:   end for
17:   counter ← counter + 1
18: until ( Converges ) or ( counter = Max )
```

It is important to notice that the second approach generates

$\#Faces \times \#cells \times (k-1)$ threads, which is bigger compared to the number of threads generated by the first approach $\#Edges \times (k-1)$, however, this does not affect negatively the performance because this algorithm is memory latency limited.

# 7. IMPLEMENTATION RESULTS

CFD is a scientific area that analyze and solve problems involving fluid flows utilizing numerical approaches. Aerospace engineering is one of the fields that led the CFD development. The application utilized in this section solves the Navier-Stokes equations on unstructured grids utilizing a high order correction procedure via reconstruction methods. Details on this method are out of the scope of this paper, for the further details refer to the paper by Wang [11].

The CFD application was originally implemented and optimized for running on CPUs. The most important optimization techniques utilized in the CPU implementation are loop unrolling, improved memory access and cache utilization.

In this section we present the results of the implementation of the CFD application on a NVidia Tesla T10 GPU utilizing the algorithm proposed in this paper. For comparison purposes we show the speedup achieved by the GPU implementation without occupancy optimization ($GPU_1$) and the one with doubled occupancy ($GPU_2$). The speedup is computed considering the original CPU implementation.

Table 3: GPU implementation of a CFD application

| **k-1** | **speedup (CPU/GPU$_1$)** | **speedup (CPU/GPU$_2$)** |
|---|---|---|
| 1 | 20 | 27 |
| 2 | 31 | 45 |
| 3 | 41 | 63 |
| 4 | 37 | 82 |
| 5 | 43 | 88 |

As it can be seen on Table 3, by doubling the occupancy, the speed is improved by at least 40% and in some cases the speedup achieved is doubled.

# 8. CONCLUSIONS

In this paper we presented the main algorithm as well as memory access patterns for applications that do analysis using unstructured grids. Implementation on a NVidia Tesla GPU of the algorithm was analyzed in terms of hardware occupancy and global memory access. This analysis led us to propose an algorithm that achieves higher occupancy and more efficient global memory access than the original algorithm. The actual GPU implementation achieved a speedup of more than 80 times compared to the CPU version.

The edge-oriented analysis was shown to be troublesome because of the random-like memory access, which linearly increased the number of memory transactions. In order to reduce the number of memory transactions, the edge-oriented analysis was transformed into a cell-oriented analysis. This new approach decreases the number of memory transactions, but at the same time increases the number of threads generated. However, this was not an issue because unstructured grid applications are memory latency limited, which means that computation is overlapped with memory operations.

# 9. ACKNOWLEDGMENTS

This manuscript has been authored by Iowa State University of Science and Technology under Contract No. DE-AC02-07CH11358 with the U.S. Department of Energy.

The authors wish to thank the Air Force Office of Scientific Research and NVIDIA Corporation for providing funds and equipment to build the GPU cluster.

We acknowledge Mark Klein at the Scalable Computing Laboratory for his efforts in setting up the GPU cluster and other support provided.

# 10. REFERENCES

[1] Owens, J.D, Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C., *GPU Computing*, Processdings of the IEEE, Vol. 95(5) pp. 879-899, 2008.

[2] NVidia, *NVIDIA CUDA Programming Guide v.2.3.1*, Aug. 2009.

[3] Owens, J.D, Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E., Purcell, T., *A Survey of General-Purpose Computation on Graphics Hardware*, Computer Graphics Forum, Vol. 26(1) pp. 80-113, Mar. 2007.

[4] Hu, H., Turner, E., *Parallel CFD Computing Using Shared Memory OpenMP*, Lecture Notes on Computer Science, pp. 1137-1146, 2001.

[5] Mavriplis, D.J., *Unstructured Grid Techniques*, Anual Review of Fluid Mechanics, Vol. 29, pp. 473-514, Jan, 1997.

[6] Kaushik, D.K., Keyes, D.E., *Efficient Parallelization of an Unstructured Grid Solver: A Memory-Centric Approach*, Istambul Technical University, 1999.

[7] Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plischker, W., Shalf, J., Williams, S., Yelick, K., *The Landscape of Parallel Computing Research: A View from Berkeley*, Electrical Engineering and Computer Sciences, University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, Dec. 18, 2006

[8] Corrigan, A., Camelli, F., Rainald, L., *Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware*, 19th AIAA Computational Fluid Dynamics, Jun, 2009.

[9] Guo, W., Jin, C., Jianhua, Li., *High performance lattice Boltzmann algorithms for fluid flows*, International Symposium on Information Science and Engineering, 2008.

[10] Nickolls, J., Dally, W., *The GPU Computing Era*, Micro, IEEE, Vol: 30,2, pp: 56-69, Mar. 2010.

[11] Wang, Z.J., Gao, H., *A unifying lifting collocation penalty formulation including the discontinuous Galerkin, spectral volume/difference methods for conservation laws on mixed grids*, Journal of Computational Physics, Vol: 228, 21, Nov. 2009.