Iran. J. Sci. Technol. Trans. Sci.
DOI 10.1007/s40995-016-0042-7

RESEARCH PAPER

Efficient Memory Access Patterns for Solving 3D Laplace Equation on GPU

Muhammad Naveed Akhtar[1] · Muhammad Hanif Durad[1] · Anila Usman[1] · Muhammad Abid Mughal[1]

Received: 7 February 2016 / Accepted: 4 June 2016

Abstract Graphic processor units (GPUs) are highly scalable parallel platforms for computation. A GPU contains thousands of cores along with different types of memory spaces having varying bandwidths. The maximum throughput of GPU computation lies in efficient use of these memory types. This paper presents research involving 12 different kernels to solve the standard Laplace equation in three dimensions. Each kernel uses a unique memory access pattern. The benchmarks have been established for the said problem and a novel efficient kernel is suggested after in-depth analysis. A throughput of more than 50 Giga floating point operations per seconds (GFLOPS) has been obtained on an average GPU as consequence of optimizing the memory access path. The best approach achieves a speedup of about 70 on the GPU in comparison to a CPU.

Keywords GPU · Laplace 3D · GPU texture · GPU surface references · GPU shared memory · Lockless synchronization · Compute unified device architecture (CUDA)

✉ Muhammad Naveed Akhtar
naveed@pieas.edu.pk

Muhammad Hanif Durad
hanif@pieas.edu.pk

Anila Usman
anila@pieas.edu.pk

Muhammad Abid Mughal
mabidm@pieas.edu.pk

[1] Department of Computer and Information Sciences, Pakistan Institute of Engineering and Applied Sciences, Islamabad, Pakistan

1 Introduction

Laplace equation is one of the elementary solution of heat transfer problems and is representative of elliptic partial differential equations (PDEs) (Michael 2002). A seven-point stencil with finite difference method (Cheney and Kincaid 2012) is used to solve this problem. The problem seems simple and easy while solving for one and two space dimensions. As the problem is extended to higher dimensions with fine grids, it can take hours to solve on the state-of-the-art CPUs (Cheney and Kincaid 2012). To speed up the progress of numerical solution, one possibility is to divide problem in multiple parts and execute them in parallel.

The NVIDIA GPUs are designed not only to render graphics, but also provide highly scalable parallel computing (Nvidia 2014b) with thousands of concurrent cores to solve computationally expensive problems. GTX GeForce 660 based on the Kepler architecture has been used to perform simulations presented in this article. The Kepler architecture is a third-generation architecture for CUDA compute applications by NVIDIA (Nvidia 2014c). Kepler uses bind-less textures which eliminate any limits on the number of textures that might be used (Nvidia 2012). CUDA architecture makes GPU well-suited for highly parallelized computations (Glaskowsky 2009; Nvidia 2011).

GPU computing is an emerging field. There exist a number of articles regarding solution and implementation of numerous problems on GPU. Chen presents a framework of GPU accelerated spectral methods for systems of coupled elliptic equations (Chen 2015). Papageorgiou and Platis present a simplification algorithm for triangular meshes on GPU (Papageorgiou and Platis 2015). Jiang et al. presented a 3D numerical parallel diffusion algorithm

Published online: 22 June 2016

in cylindrical coordinates on GPU devices (Jiang et al. 2015).

Only a few research articles provide benchmark results specific to Laplace and Poisson problems. Unat et al. presented the performance of the 3D stencil method on their proposed programming model called "Mint for GPU" (Unat et al. 2011). Dugan, Genovese and Goedecker benchmarked analysis results regarding different boundary conditions for 3D Poisson equation on GPU (Dugan et al. 2013). A red/black SOR variant for GPU is presented by Konstantinidis and Cotronis (Konstantinidis and Cotronis 2013). Helfenstein and Koko implemented a parallel version of preconditioned conjugate gradient algorithm on a GPU platform (Helfenstein and Koko 2012).

In this paper, a detailed analysis of efficient use of available memory spaces on GPU has been presented and simulations have been performed for each case. The widely used heat transfer problem called Laplace equation, extended for 3-dimensions is used to exploit various possible memory access strategies on GPU. An in-depth analysis has been carried out regarding timing, cache, warp divergence and memory throughput measurements. A total of 12 GPU kernels have been simulated for the purpose; each includes a unique methodology to exploit the memory spaces of GPU for single precision calculations. The focus of this work is to determine the best way to utilize GPU memory spaces to solve a computationally intensive problem on GPU.

2 Laplace Formulation in 3D

Standard Laplace equation is given as:

$$\nabla^2 w = 0 \tag{1}$$

In three space dimensions:

$$\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} = 0 \tag{2}$$

Model heat problem (Cheney and Kincaid 2012) is extended for unit cube with these boundary conditions given in equation set (3).

$$\begin{aligned} w(0,y,z) &= w(x,0,z) = w(x,y,0) = 0 \\ w(1,y,z) &= sin(\pi yz) \\ w(x,1,z) &= sin(\pi xz) \\ w(x,y,1) &= sin(\pi xy) \end{aligned} \tag{3}$$

Taylor series expansion for 2nd order finite difference in one dimension is given in Eq. (4) below

$$f''(x) \approx \frac{1}{h^2}[f(x-h) - 2f(x) + f(x+h)] \tag{4}$$

Incorporating above Eq. (4) in 3D Laplace Eq. (2), it becomes as:

$$\begin{aligned} \nabla^2 w \approx \frac{1}{h^2}[&w(x+h,y,z) + w(x-h,y,z) + w(x,y+h,z) \\ &+ w(x,y-h,z) + w(x,y,z+h) + w(x,y,z-h) \\ &- 6w(x,y,z)] = 0 \end{aligned} \tag{5}$$

By discretization of problem it yields as:

$$w_{i,j,k} = \frac{1}{6}[w_{i+1,j,k} + w_{i-1,j,k} + w_{i,j+1,k} + w_{i,j-1,k} + w_{i,j,k+1} + w_{i,j,k-1}] \tag{6}$$

This formula provides an approximation of order $O(h^2)$. As the grid spacing goes narrower and smaller, the error approaches zero rapidly. This is 7-point formula; it involves the 6 nearest grid points in 3D to solve one $w(x, y, z)$ at middle point of stencil. The arrangement is shown in Fig. 1.

As we decrease h to make grid fine for better accuracy, the computational complexity rises as $O(n^3)$; even for $h = 0.01$ for a unit cube the number of grid point reaches a million. Current state-of-the-art CPU may take hours to reach the solution as the problem size increases. The number of iterations needed to reach the solution also increases exponentially (Cheney and Kincaid 2012).

3 GPU Implementation

As evident from Eq. (6) in Sect. 2, each point computation involves six memory read and one memory write operation. The memory I/O considerably limits GPU performance. Two main 3-dimensional arrays named as $A1$ and $A2$ are used to solve the stated problem. These arrays serve as source and destination for temperature calculations. Each element of these arrays corresponds to one point of 3D unit cube in this problem. Six arrays in 2D space named as $B1, B2, \ldots, B6$ are used to contain boundary temperature on each side of cube. A total of 12 GPU kernels have been

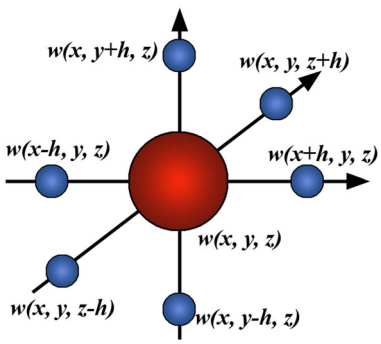

Fig. 1 Seven-point Laplace stencil in 3 space dimensions

simulated using different memory access techniques. These techniques vary from a naive approach to a highly efficient way to deal with this problem. 3D Block partitioning and these kernels are explained briefly as under.

3.1 3D Block Partitioning

For this problem, we have used 3D block partitioning which is an extension of 2D data distribution (Kumar et al. 1994) for both source and destination arrays. Each block consists of $10 \times 10 \times 10$ elements being handled by separate threads. The size of a block is limited by the maximum number of threads it may contain. For our target architecture, this number is 1024. The threads, blocks and data partitioning is explained in Fig. 2. All other boundary conditions are stored using 2-dimensional arrays on GPU.

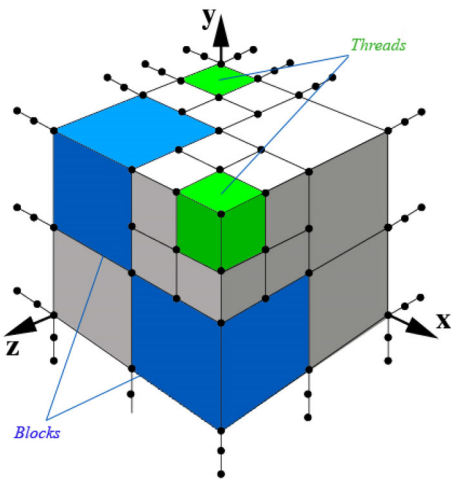

Fig. 2 Memory distribution of 3D array on GPU threads and blocks

3.2 Kernel K1

This kernel uses both main arrays $A1$, $A2$ and all the six boundary condition arrays $B1, \cdots, B6$ from global memory of GPU. It uses 7-point stencil to solve a point in 3D, i.e., right, left, up, down, front back and middle. 3D block distribution for both these arrays is used in this kernel. After each iteration the blocks are synchronized on CPU (Nvidia 2014a), source and destination array pointers are swapped and the process continues. A snapshot for one point of this kernel calculation is given in Fig. 3.

3.3 Kernel K2

The boundary conditions $B1, \cdots, B6$ are read-only arrays for this problem. GPU textures are read-only memory locations which may be efficiently used to read these arrays. This kernel uses these boundary arrays from GPU texture memory. All other parameters are the same as that in kernel K1. The process is shown in Fig. 4.

3.4 Kernel K3

This kernel exploits shared memory on GPU. An array of size $n \times n \times n$ requires $6n^3$ memory elements to be solved. With careful observation we can see that there are about $5n^3$ repetitions of array elements in this computation. We can reduce this overhead of global memory calls by first copying the relevant data in a shared array inside each block. Each thread of GPU initially loads one element from global memory into the shared memory. Then the entire computation for each block is carried out using elements from the shared memory except for boundary elements which require an additional element from global memory. The block synchronization methodology for this kernel is

Fig. 3 Listing of kernel for solving Laplace 3D (K1)

```
__global__ void Laplace3D_K1 ()
{
    int i = blockIdx.x * blockDim.x + threadIdx.x ;
    int j = blockIdx.y * blockDim.y + threadIdx.y ;
    int k = blockIdx.z * blockDim.z + threadIdx.z ;
    float value = 0.0;
    if((i<size) && (j<size) && (k<size))
    {
        //Add left cell value (from Boundary or input)
        if (i == 0) value += b_left[j*size + k]; // Left Boundary
        else value += input[(i-1)*size*size + j*size + k ];

        //Similarly Add 5 other points (right, up, down, top, bottom)
        ...

        output[i*size*size + j*size + k] = value/6;
    }
}
```

Fig. 4 Listing of kernel for solving Laplace 3D (K2)

```
__global__ void Laplace3D_K2 ()
{
    ...
    if((i<size) && (j<size) && (k<size))
    {
        //Add left cell value (LeftBoundary from texture)
        if (i == 0) value += tex2D(tx2_left, j, k);
        else value += input[(i-1)*size*size + j*size + k ];

        //Similarly add 5 other points (right, up, down, top, bottom)
        ...

        output[i*size*size + j*size + k] = value/6;
    }
}
```

Fig. 5 Listing of kernel for solving Laplace 3D (K3)

```
__global__ void Laplace3D_K3 ()
{
    __shared__ float shared_array[10][10][10];
    ...
    if((i<size) && (j<size) && (k<size))
    {
        shared_array[threadIdx.x][threadIdx.y][threadIdx.z] = input[...];
        __syncthreads();
        //Add left cell value
        if (i == 0)                 // Left boundary condition
            value += b_left[j*size + k];
        else if(threadIdx.x == 0)   // Cell on left edge of Block
            value += input[(i-1)*size*size + j*size + k ];
        else                        // Get data from shared memory
            value += shared_array[threadIdx.x-1][threadIdx.y][threadIdx.z];

        // Similarly add 5 other points here

        output[i*size*size + j*size + k] = value/6;
    }
}
```

same as that of K1. The kernel procedure is explained in Fig. 5.

3.5 Kernel K4

This kernel uses advantages of both K2 and K3 simultaneously. It exploits shared memory for computation as well as uses GPU texture cache to read all six boundary conditions. Again block synchronization is implicitly performed by the CPU as explained in K1. The kernel is listed in Fig. 6.

3.6 Kernel K5

This kernel uses a non-conventional way to read and write both main arrays. $A1$,$A2$ are treated as read-only and as output alternatively due to the nature of problem. This kernel exploits textures and surfaces for these arrays. $A1$ is treated as a texture array and $A2$ is treated as a surface reference when $A2$ is being updated and $A1$ is read-only. Similarly for the next iteration $A2$ is treated as texture array and $A1$ is treated as surface reference. One such iteration is explained in Fig. 7. All the six boundary arrays being read-only are treated as textures and CPU block synchronization has been used once again.

3.7 Kernel K6

This kernel uses both $A1$ and $A2$ as surface reference on GPU. The usage of textures is avoided to understand the performance difference of both textures and surfaces. All the remaining setup is the same as that of K5. One step of this kernel computation is given in Fig. 8.

3.8 Kernel K7

This kernel tries to exploit shared memory on GPU along with surface references. The data are first fetched into shared memory of each block from the input array. Then, all the computation is carried out locally. Finally, the results are exported to the output array using surface references. Remaining setup is the same as that of mentioned in K5. Details of this kernel are shown in Fig. 9.

Fig. 6 Listing of kernel for solving Laplace 3D (K4)

```
__global__ void Laplace3D_K4 ()
{
    ...
    if((i<size) && (j<size) && (k<size))
    {
        shared_array[threadIdx.x][threadIdx.y][threadIdx.z] = input[...];
        __syncthreads();
        //Add left cell value
        if (i == 0)                 // Left boundary from texture
            value += tex2D(tx2_left, j, k);
        else if(threadIdx.x == 0)   // Cell on left Edge of Block
            value += input[(i-1)*size*size + j*size + k ];
        else                        // Get data from shared memory
            value += shared_array[threadIdx.x-1][threadIdx.y][threadIdx.z];

        // Similarly add 5 other points here

        output[i*size*size + j*size + k] = value/6;
    }
}
```

Fig. 7 Listing of kernel for solving Laplace 3D (K5)

```
texture <float, cudaTextureType3D, cudaReadModeElementType> tx_input
surface <float, cudaSurfaceType3D> surf_output

__global__ void Laplace3D_K5 ()
{
    ...
    if((i<size) && (j<size) && (k<size))
    {
        //Add left cell value
        if (i == 0) value += tex2D(tx2_left, j, k);
        else value += tex3D(tx_input, i-1, j, k);

        //Similarly add 5 other points here

        surf3Dwrite(value/6.0, surf_output, i * sizeof(float), j, k);
    }
}
```

Fig. 8 Listing of kernel for solving Laplace 3D (K6)

```
surface <float, cudaSurfaceType3D> surf_input
surface <float, cudaSurfaceType3D> surf_output

__global__ void Laplace3D_K6 ()
{
    ...
    if((i<size) && (j<size) && (k<size))
    {
        //Add left cell value
        if (i == 0)
            value += tex2D(tx2_left, j, k);
        else {
            surf3Dread(&v, surf_input, (i-1) * sizeof(float), j, k);
            value+=v;
        }

        //Similarly add 5 other points here

        surf3Dwrite(value/6.0, surf_output, i * sizeof(float), j, k);
    }
}
```

Fig. 9 Listing of kernel for solving Laplace 3D (K7)

```
surface<float, cudaSurfaceType3D> surf_input;
surface<float, cudaSurfaceType3D> surf_output;

__global__ void Laplace3D_K7 ()
{
    __shared__ float shared_array[10][10][10];
    ...
    if((i<size) && (j<size) && (k<size))
    {
        surf3Dread(&v, surf_input, i * sizeof(float), j, k);
        shared_array[threadIdx.x][threadIdx.y][threadIdx.z] = v;
        __syncthreads();
        //Add left cell value
        if (i == 0)
            value += tex2D(tx2_left, j, k);
        else if(threadIdx.x == 0) {
            surf3Dread(&v, surf_input, (i-1) * sizeof(float), j, k);
            value+=v;
        }
        else
            value += shared_array[threadIdx.x-1][threadIdx.y][threadIdx.z];

        // Similarly add 5 other points here

        surf3Dwrite(value/6.0, surf_output, i * sizeof(float), j, k);
    }
}
```

3.9 Kernel K8

This kernel uses a sparse system matrix in diagonal format and a vector of initial guess to start with. It uses Jacobi method to calculate error and repeatedly performs matrix–vector dot product. This method requires about three times more memory than that of the kernel specified earlier. The memory requirement increased due to involvement of the system matrix. This method is presented in Jost, Contassot-Vivier, and Vialle.

3.10 Kernel K1a,···, K4a

Kernels K1a to K4a have the same memory arrangement as that of K1 to K4, respectively. The only difference is that these kernels use lockless synchronization in between iterations as presented in Xiao and Feng (2010). The key concept is to iterate inside the GPU and issue a synchronization call before the next iteration. This procedure is explained in Fig. 10.

4 Target Architecture

NVIDIA GeForce 660 GPU having compute capability of 3.0 was used to carry out all the simulations presented in this article. This GPU has 2 GB of global memory and 960 CUDA enabled cores with a clock rate of 980 MHz. It can execute 10,240 threads, not exceeding 1024 threads per block. All these threads are executed concurrently by 5 Streaming Multiprocessors (SM) inside GPU each capable of running 2048 threads.

The GPU results were compared with 5th Generation Intel® Core™ i5 Processor with a clock rate of 2.67 GHz bearing 8 GB of DDR3 Random Access Memory (RAM) by executing the same algorithm implemented using GCC. NVIDIA CUDA Toolkit 6.5 and Linux (Fedora 20) operating system was used to execute all the GPU simulations.

4.1 GPU Memory Overview

NVIDIA GPU have multiple types of memory, each categorized by different bandwidth and latency (Nvidia). This memory hierarchy includes shared memory, local memory, registers, global memory, texture memory and constant memory. Global memory having the highest latency is the largest memory accessible by both device (GPU) and host (CPU); responsible for data transfer between them. Constant memory and texture memory has the same latency as global memory, but these are read-only for threads, whereas registers, local memory and shared memory are much faster with relatively limited space (Gray et al. 2013). These six memory spaces are illustrated in Fig. 11. GPUs work under the "Stream" model of computation (Gray et al. 2013).

5 Results and Discussion

Steady-state temperature profile of the unit cube obtained after 32k iterations on GPU is given in Fig. 12. At 32k iterations the sum of squares of error called epsilon (ε) between 2 consecutive iterations was of order 1e-16. The

Fig. 10 Lockless synchronization methodology (Xiao and Feng)

```
__device__ Perform_Laplace_iteration(float *input, float *output);
__device__ GPU_Lockless_Sync();

__global__ Laplace_main(float *input, float *output)
{
    For(int i=0; i< max_iterations)
    {
        Perform_Laplace_iteration(input, output);
        GPU_Lockless_Sync(); //Lock-less synchronization
        swap(input, output);
    }
}
```

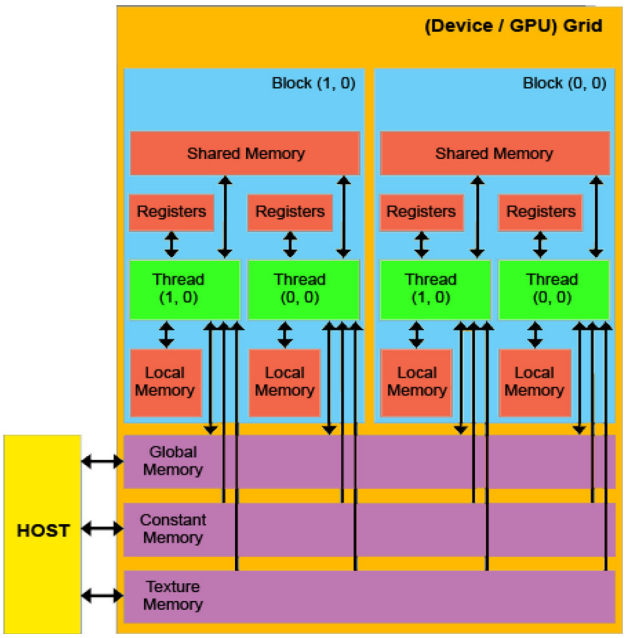

Fig. 11 Overview of available memory spaces on GPU

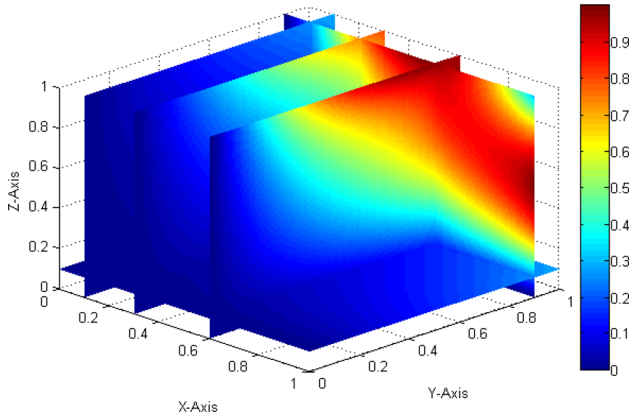

Fig. 12 Steady-state temperature distribution on unit cube after 32k iterations

GPU and CPU results were compared and found to be identical up to 10 decimal places. A few representative values corresponding to some sample points from CPU and GPU are presented in Table 1. The small difference is because of the way the floating points are approximated on CPU and GPU and order of operation on floats (Whitehead and Fit-Florea 2011). In-depth analysis is performed to determine the efficiency of kernels stated in Sect. 3. The analyses include but not limited to CPU/GPU time taken, computation overheads, warp divergence, different memory types' efficiency and cache throughputs.

The lockless synchronization only degrades performance. The method presented in Xiao & Feng forces to execute no more than two blocks simultaneously. It affects the scalability of parallelism of problem hence the performance got worse. That is the reason that kernels $K1a, \ldots, K4a$ are not included in the results here. Similarly K8 demands about three times more memory for storage of system matrix (Jost et al. 2009). It uses sparse multiplication as an added overhead for GPU. This approach takes considerable increased time and resources of GPU. Hence kernels $K1, \ldots, K7$ are the most suitable to be presented in this section for comparison on same ground.

Problem size was varied from 10 to 500 in each dimension. A unit cube is considered in all simulations having equal space steps h in all three dimensions. Initially the analysis was carried out for a single iteration and then it was extended to 1000 iterations to incorporate latency of GPU block synchronization on CPU. NVIDIA profiler was used to obtain these results. The simulations were repeated multiple times on GPU and average results are reported in this section.

5.1 Timing Analysis

Figures 13 and 14 provide an overall picture of the kernels with respect to the wall time taken by all these kernels for a single iteration. The most prominent result is that all these kernels on GPU are taking considerably less time to solve the problem than CPU. It is also evident that K5 performs the best and takes minimum computation time in comparison to all other kernels. Figure 14 shows that memory

Table 1 Comparative accuracy of selected results from GPU with CPU data as reference

x	y	z	CPU values	GPU values	Absolute difference
0.2	0.2	0.2	0.017118069375724	0.017118069178895	1.97E − 10
0.2	0.2	0.8	0.076174559404692	0.076174559181386	2.23E − 10
0.2	0.8	0.2	0.076174559404692	0.076174559181386	2.23E − 10
0.2	0.8	0.8	0.309755952985546	0.309755952732202	2.53E − 10
0.8	0.2	0.2	0.076174559404692	0.076174559181386	2.23E − 10
0.8	0.2	0.8	0.309755952985546	0.309755952732202	2.53E − 10
0.8	0.8	0.2	0.309755952985546	0.309755952732202	2.53E − 10
0.8	0.8	0.8	0.816442794091899	0.816442793804475	2.87E − 10

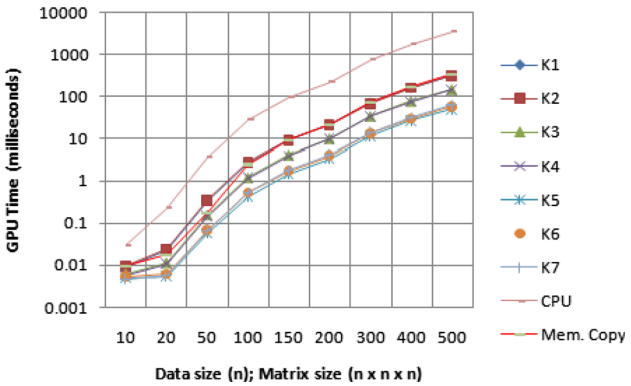

Fig. 13 Computation time comparison for GPU kernels and CPU

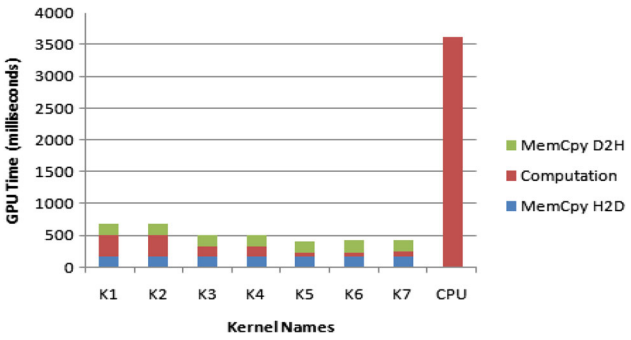

Fig. 14 Computation time and memory transfer time for GPU kernels

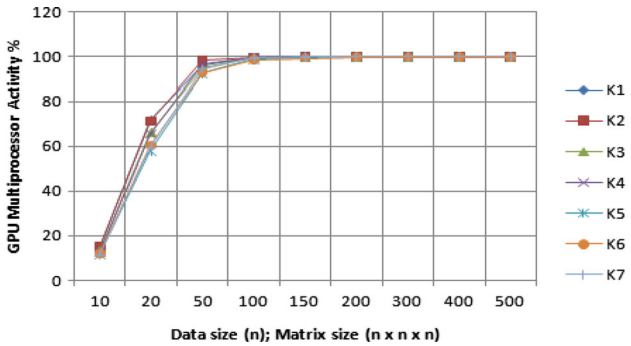

Fig. 15 GPU streaming multiprocessors activity analysis for all kernels

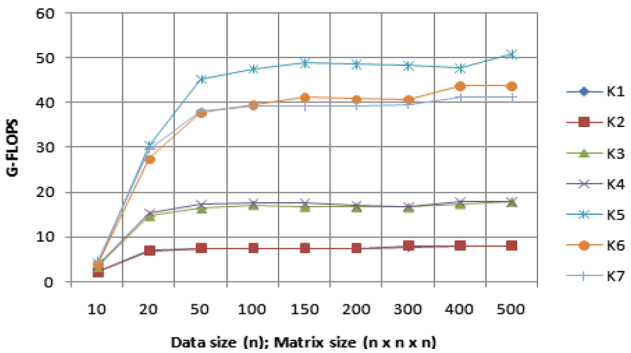

Fig. 16 Comparison of G-FLOPS achieved for all kernels on GPU

copying from host to device (H2D) and from device to host (D2H) takes considerably more time than the computation itself. However, even if one includes this memory transfer time, it yields better results than those from the CPU. Memory transfer time is approximately the same for all the kernels.

5.2 GPU Multi-Processor Activity

Figures 15 and 16 show streaming multiprocessor (SM) activity on GPU. Figure 15 illustrates that all kernels achieve about 100 % utilization of SM as the input data size is increased. However, Fig. 16 depicts that K5 provides the greatest computational throughput. The efficient memory arrangement in case of K5 was the reason to achieve nearly 51 GFLOPS computational throughput from GPU. As stated earlier that each thread requires seven memory operations to perform a single computation, keeping this in view 51 GFLOPS is a considerably high number. In contrast kernels K1, K2 were barely able to get computational throughput of 8 GFLOPS. This is the reason that K5 is also taking lesser time to complete the task as compared to all other kernels.

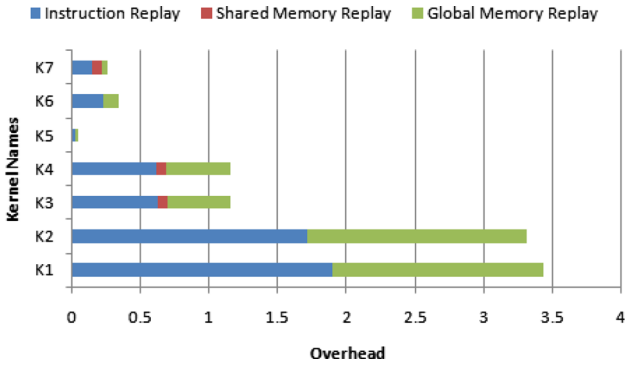

Fig. 17 Comparison of GPU overheads for each kernel

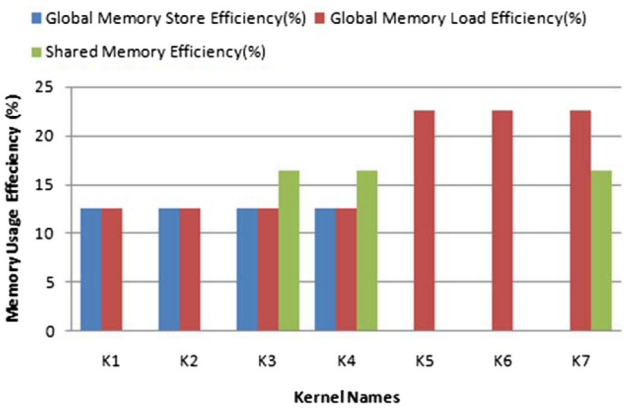

Fig. 18 GPU memory usage affiance for each kernel

Fig. 19 GPU cache hit rate achieved by each kernel

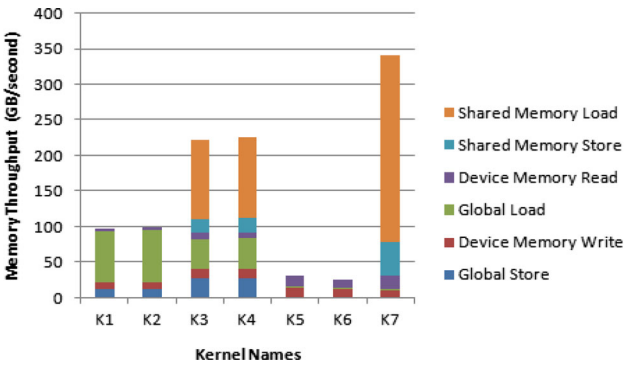

Fig. 20 GPU memory throughputs achieved by each kernel

5.3 Overheads

By using both textures and surfaces alternatively in case of K5, we were able to reduce computation overhead to its lowest. Figure 17 summarizes the overheads (Cheng et al. 2014) faced by GPU while executing each kernel. K5 has only instruction and a little global memory replay overheads. Shared memory replay overhead is absent in K1, K2, K5 and K6, as they are not using shared memory. As a consequence of minimum overhead, K5 beats all peers.

5.4 Memory and Cache Use Efficiency

Figures 18 and 19 provide another evidence for the best performance of K5. K5, K6 and K7 using surface references have memory load efficiency (Cheng et al. 2014) higher than K1–K4.

Caches are the fastest memory locations. Figure 19 depicts that K5 has the maximum cache hit rate for all 3 caches, i.e., L2, L1 and texture. It gets most of the data required from these caches. K2 and K4 also use all three caches, but not as efficiently as K5.

5.5 Memory Throughputs

Figures 20 and 21 indicate that K5 has maximum memory throughput for cache read/writes and bears minimum memory throughput for low-speed memories like global and device memory read/writes. Again it is evident that K5 obtains maximum data from all three caches of GPU and at a higher speed (smarter cache reuse) than those of other peer kernels. Figure 22 supports the claim that K5 issues the minimum number of data load transaction requests from global memory. Due to the L1 cache being not supported for global memory transactions in GTX-660, each memory transaction can fetch anywhere from 32 bytes to 128 bytes as a multiple of 32 bytes (Cheng et al. 2014). This information, together with the number of transactions, can be used to calculate the minimum and maximum global memory accesses in MBs.

5.6 Timing Analysis for 1000 Iterations

Figure 23 explains that initially memory copying time is considerably higher than computational time. It also indicates that as the number of iterations is increased to obtain

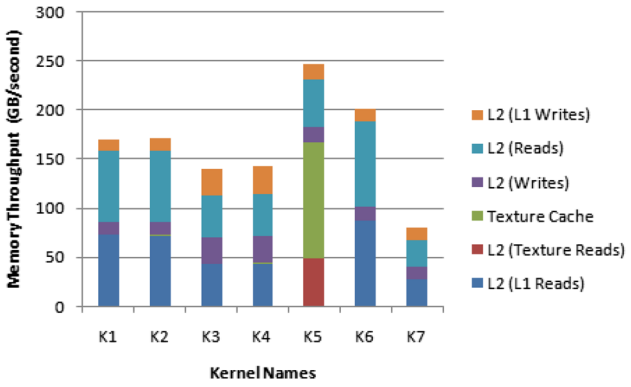

Fig. 21 GPU cache throughputs achieved by each kernel

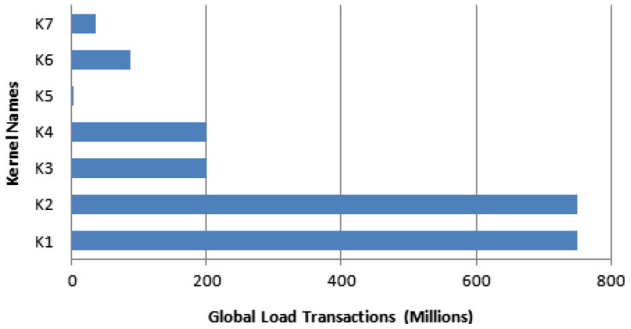

Fig. 22 Data load transactions issued by each kernel from global memory

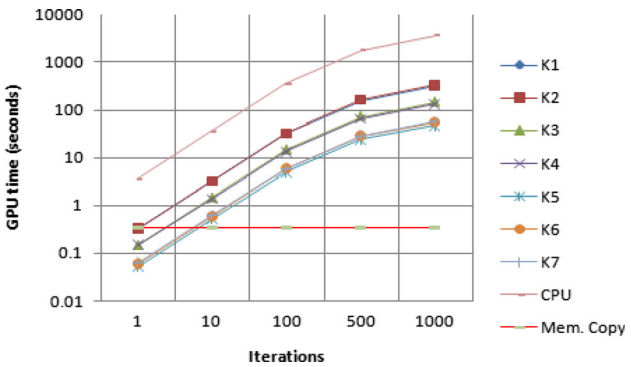

Fig. 23 Comparison of time taken for 1000 Laplace iteration

higher accuracy, this memory transfer time becomes negligible and may be ignored.

5.7 Speedup

From Fig. 24 it is evident that the K5's efficient use of memory provides the highest throughput and therefore it provides a speedup (Kumar et al. 1994) of 70 than CPU. The next best approach is to use the memory arrangement presented in K6; it is about 60 times faster than average CPUs.

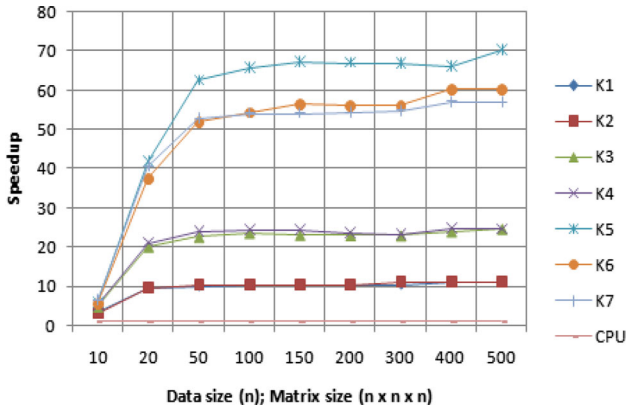

Fig. 24 Speedup achieved in case of each kernel execution

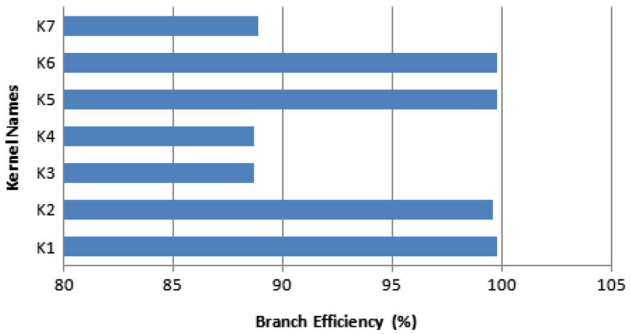

Fig. 25 Branch efficiency (%) for each kernel

5.8 Branch Efficiency

The boundary threads, which are about 1.2 % of total threads for cube dimensions of 500 × 500 × 500, get data from boundary arrays. The use of different arrays for boundary positions could potentially lead to divergent code paths. This investigation leads to analysis regarding branch efficiency for warp divergence. The same is carried out and presented in Fig. 25. In the worst approach the warp divergence was at most 12 % and the best approach restricted divergent warps to less than 1 %.

6 Conclusion and Future Work

The best throughput from GPU lies in efficient use of available memory spaces in GPU. The bottleneck of GPU computation lies in memory bandwidth and not in processing power. It has been shown using different kernels that computational performance may degrade if memory spaces are not accessed properly. The kernel K5 presented is the fastest kernel for solving Laplace equation using 7-point stencil method since it uses textures and surface reference as much as possible in a unique way. Therefore,

use of the textures and surface references is highly recommended while programming for GPU rather than conventional memory access methods where possible. The same kernel gained a speedup of 70 and more than 50 GFLOPS GPU throughput over an average CPU by utilizing memory, caches and registers efficiently.

We aim to extend this work by solving and benchmarking results of other problems involving partial differential equations on GPU.

References

Gray A, Sjöström A, llieva-Litova N (2013) Best Practice mini-guide accelerated clusters. Using General Purpose GPUs

Chen F (2015) A new framework of GPU-accelerated spectral solvers: collocation and Glerkin methods for systems of coupled elliptic equations. J Sci Comput 62(2):575–600

Cheney E, Kincaid D (2012) Numerical mathematics and computing. Nelson Education

Cheng J, Grossman M, McKercher T (2014) Professional Cuda C Programming. Wiley

Dugan N, Genovese L, Goedecker S (2013) A customized 3D GPU Poisson solver for free boundary conditions. Comput Phys Commun 184(8):1815–1820

Glaskowsky PN (2009) NVIDIA's Fermi: the first complete GPU computing architecture. White paper

Helfenstein R, Koko J (2012) Parallel preconditioned conjugate gradient algorithm on GPU. J Comput Appl Math 236(15):3584–3590

Jiang B, Dai W, Khaliq A, Carey M, Zhou X, Zhang L (2015) Novel 3D GPU based numerical parallel diffusion algorithms in cylindrical coordinates for health care simulation. Math Comput Simul 109:1–19

Jost T, Contassot-Vivier S, Vialle S (2009) An efficient multi-algorithms sparse linear solver for GPUs. Paper presented at the ParCo

Konstantinidis E, Cotronis Y (2013) Graphics processing unit acceleration of the red/black SOR method. Concurr ComputPract Exp 25(8):1107–1120

Kumar V, Grama A, Gupta A, Karypis G (1994) Introduction to parallel computing: design and analysis of algorithms. Benjamin/Cummings Publishing Company, Redwood City

Michael TH (2002) Scientific computing: an introductory survey. The McGraw-Hill Companies Inc., New York

Nvidia (2011) Tuning CUDA Applications for fermi version 1.0. NVIDIA, May

Nvidia (2012) NVIDIA GeForce GTX 680 Whitepaper: NVIDIA Corporation

Nvidia (2014a) CUDA C Best Practices Guide version 6.5

Nvidia (2014b) CUDA C programming guide version 6.5. NVIDIA Corporation, Santa Clara

Nvidia (2014c) Tuning CUDA applications for Kepler

Papageorgiou A, Platis N (2015) Triangular mesh simplification on the GPU. Vis Comp 31(2):235–244

Unat D, Cai X, Baden SB (2011) Mint: realizing CUDA performance in 3D stencil methods with annotated C. In: Paper presented at the Proceedings of the international conference on Supercomputing

Whitehead N, Fit-Florea A (2011) Precision and performance: floating point and IEEE 754 compliance for NVIDIA GPUs. rn (A + B), 21:1–1874919424

Xiao S, Feng WC (2010) Inter-block GPU communication via fast barrier synchronization. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium, IEEE, pp 1–12