Neural, Parallel, and Scientific Computations 21 (2013) 361-374

GPU METRICS FOR A LINEAR SOLVER

M. PANCHATCHARAM[1], S. SUNDAR[2], AND A. KLAR[3]

[1]Fraunhofer ITWM, Kaiserslautern, Germany
[2] Department of Mathematics, IIT Madras, Chennai
[3]Fachbereich Mathematik, TU Kaiserslautern, Germany

ABSTRACT. GPU metrics is a set of measurements used to distinguish an efficient GPU and GPU algorithms. In this paper, we present a mathematical definition of three important GPU metrics such as occupancy, average bandwidth utilization and volume in order to prove certain preliminary results. We discuss another most important and frequently used GPU metric called acceleration ratio. Finally, we discuss the importance and application of GPU metrics on BICGStab algorithm.

AMS (MOS) Subject Classification. 65F10, 65F30, 65F99

1. Introduction

Graphics Processing Units (GPUs) [1] developed to manipulate computer graphics in more efficient and effective way, nowadays become widely useful in parallel computing, as they process large blocks of data in parallel. Advanced embedding chipsets with GPU in modern mobiles and smartphones are capable of multitasking due to their efficient programming. As GPUs are essential part of those chipsets, the GPU performance is becoming most important. Due to the rapid and enormous growth in graphics applications such as computer games and simulations using GPU, expensive and time consuming super computing such as peak performance near teraflop are available for hundreds of dollars to a few thousand dollars. As the cost per gigaflop is very low in GPU, it has motivated the researchers to utilize it in an efficient non-graphics applications such as solving large linear systems, weather prediction, data mining and so on. Most of the leading software companies claims that GPU is the best tool for the future. But, one has to look after programming of the application code in order to achieve performance near the peak.

GPU programming is little complicated as it has different memories such as device memory, shared memory, global memory, constant cache, texture cache and registers, each with different latency. Each streaming processor in a GPU consists of a partitioned scalar processors or cores. Each GPU has different capability which can be measured using their metrics. Although, a well written algorithm for a software in GPU should work independent of the choice of GPU, one has to study in detail GPU

Received July 15, 2013

metrics to obtain the peak performance. In this paper, we mathematically define GPU metrics in order to mathematically claim certain important results.

In the following sections, mathematical definitions of GPU metrics are introduced. Following these definitions, matrix-matrix and matrix-vector multiplication algorithm on GPU is discussed. In the final section, we present the results which include GPU metrics for linear solvers such as BICGStab.

2. **GPU metrics**

Although, the most important keywords related to GPU can be found in [2], we state them for the matter of convenience.

The smallest unit of parallelism in CUDA is the **thread**, a small piece of concurrent code with associated state that will be executed physically in parallel with other threads on the device. However, when compared to threads on CPU (host), GPU (device) threads have much lower resource usage and lower creation and switching cost (GigaThread Architecture). GPUs are only effective when running a high number of such threads. A (thread) **block** is a group of threads that can communicate with each other and synchronise their execution. Each thread in a block can be identified by its thread index. For convenience, CUDA allows the specification of the **thread id** as a scalar, two dimensional or three dimensional value. The maximum number of threads per block is limited by the hardware. A group of blocks that have same dimensionality and execute the same CUDA program logically in parallel is called a **grid**. Each block in a grid is identified by its unique *block index*, again specifiable as **block id** in scalar, 2D or 3D fashion.

Each block is being split in SIMD groups called **warps**. This splitting is guaranteed to be in a linear, consecutive fashion, with the thread with thread index 0 being in the first warp. However, the order of execution of warps is not specified. The number of threads per warp is called the **warp size**, a hardware dependant constant. A **half warp** is defined to be either the first or the second half of a warp. In other words: a warp is a group of threads executed physically in parallel.

Every multiprocessor contains a set of 32 or 64 bit **registers** per processor, fast on-chip memory that is shared by all processors on that multiprocessor called **shared memory**, accessible as fast as the register file if certain conditions are met, a **constant cache**, optimised for 1D locality, and a **texture cache**, optimised for 2D spatial locality. Constant memory and texture memory are implemented as special regions in device memory, accessible via its respective cache.

2.1. **Occupancy.**

One should try to use as much of the existing hardware as possible to achieve optimal performance. For instance, as all multiprocessors execute a whole warp in a given time frame, the number of threads per block should be a multiple of the warpsize to avoid wasting computational resources in under-populated warps. An useful metric in this context is called occupancy, defined as follows:

Definition 2.1 (Occupancy). The number of warps running concurrently on a single multiprocessor (MP) divided by the number of warps that could be run concurrently is called (multiprocessor) occupancy.

That is,

$$occupancy = \frac{Number\ of\ warps\ co-scheduled\ per\ SM}{Maximum\ Number\ of\ warps\ per\ SM} \tag{2.1}$$

The number of warps co-scheduled on a multiprocessor is determined by several factors. For example, the occupancy of a multiprocessor varies from one kernel to the next. Suppose, the GPU and kernel are fixed, the occupancy varies with the block size (i.e., number of threads per block) with which the kernel is invoked. Moreover, the same kernel and block size may result in different occupancy values on different GPUs.

Let us find the occupancy equation using the GPU factors. Here, $\gamma, \zeta, \mu, \eta, \alpha$ and ξ denote warp, block, thread, register, shared memory and memory respectively.

To find the occupancy (λ) of a kernel, we have to find γ_{SM} (number of warps that will be co-scheduled on a streaming multiprocessor), which is given by

$$\gamma_{SM} = \zeta_{SM} \times \gamma_{PB}, \tag{2.2}$$

where ζ_{SM} denotes the number of blocks co-scheduled on an SM and γ_{PB} denotes warps per block.

Let us define few more terms to compute occupancy.

$$\gamma_{PB} = \lceil \frac{\mu_{reqB}}{\phi} \rceil \tag{2.3}$$

$$\gamma_G = \lfloor \frac{\gamma_{PB}}{\gamma_{AU}} \rfloor \times \gamma_{AU} \tag{2.4}$$

$$\eta_{PB} = \begin{cases} \lceil \frac{\gamma_{reqT} \times \phi}{\eta_U} \rceil \times \eta_U \times \gamma_G & \text{for GPU1} \\ \lceil \frac{\gamma_G \times \gamma_{reqT} \times \phi}{\eta_U} \rceil \times \eta_U & \text{for GPU2} \end{cases}$$

$$\xi_{PB} = \lceil \frac{\alpha_{PB}}{\alpha_U} \rceil \times \alpha_U \tag{2.5}$$

$$\zeta_{\lim}|_\gamma = \min\{\zeta_{\max}, \lfloor \frac{\gamma_{\max}}{\gamma_{PB}} \rfloor\} \tag{2.6}$$

$$\zeta_{\lim}|_\eta = \begin{cases} 0 & \text{if} \quad \gamma_{reqT} > \eta_{T\max} \\ \lfloor \frac{\eta}{\eta_{PB}} \rfloor & \text{if} \quad 0 < \gamma_{reqT} \le \eta_{T\max} \\ \zeta_{\max} & \text{else} \end{cases}$$

$$\zeta_{\lim}|_\alpha = \begin{cases} \lfloor \frac{\alpha}{\xi_{PB}} \rfloor & \text{if} \quad \alpha_{PB} > 0 \\ \zeta_{\max} & \text{else} \end{cases}$$

$$\psi = \min\{\zeta_{\text{lim}}|_{\alpha}, \zeta_{\text{lim}}|_{\eta}, \zeta_{\text{lim}}|_{\gamma}\} \tag{2.7}$$

$$\lambda = \frac{\gamma_{PB} \times \psi}{\gamma_{\max}} \tag{2.8}$$

Fore more details about notations, please refer the glossary and occupancy calculator [3].

From (2.8), it is clear that λ is a function of $\xi_{\max}, \gamma_{\max}, \mu_{reqB}, \phi, \eta, \gamma_{reqT}, \eta_U, \gamma_{AU}, \alpha, \alpha_{PB}, \alpha_U$. For any fixed GPU, all parameters other than $\mu_{reqB}, \gamma_{reqT}, \alpha_{PB}$ are fixed. Hence λ is a function of $\mu_{reqB}, \gamma_{reqT}$ and α_{PB} for a fixed GPU.

Proposition 2.2. $0 \le \lambda \le 1$ for any GPU.

Proof. Obviously λ cannot be negative. Suppose, $\lambda > 1$, then

$$\frac{\gamma_{\max}}{\gamma_{PB}} < \psi \le \zeta_{\text{lim}}|_{\gamma} \le \lfloor \frac{\gamma_{\max}}{\gamma_{PB}} \rfloor,$$

which is a contradiction. $\square$

2.2. **Average Bandwidth Utilization and Volume.**

Let us define two more metrics namely *average bandwidth utilization* and *Volume* which are important to study GPU performance.

Average bandwidth utilization (β) is an important metric with respect to device-memory transactions.

Definition 2.3 (Average bandwidth utilization (β)). Suppose $a_1, a_2, \cdots, a_m$ are m data required to compute a. Then, average bandwidth utilization is defined as

$$\beta = \frac{1}{m}\sum_{i=1}^{m} \delta_i \tag{2.9}$$

where δ_i denotes the ratio between number of bytes used for a_i and transaction size of a_i.

Now, let us look at the definition of another important metric called volume.

Definition 2.4 (Volume). The *volume* of data transfer between the SMs and device memory is the sum of the number of transactions of each multiplied by the transaction size, i.e.,

$$V = \sum_{i=1}^{m} N_i * T_i \tag{2.10}$$

where N_i and T_i denote the number of transactions and transaction size for a_i respectively.

Data volume divided by the bandwidth between device memory and the SMs is a lower bound on the time spent transferring data between device memory and the SMs. This is also a lower bound on the time for the entire computation. So, it often plays to reduce the volume of data transfer. More details of β and V can be found in next section.

To test the acceleration performance, an acceleration ratio (GPU-speed up) $\mathfrak{A}$ is defined as

$$\mathfrak{A} = \frac{t_{CPU}}{t_{GPU}} \tag{2.11}$$

where the total processing time on CPU, t_{CPU}, comprises only the time of main program executed while the total processing time on GPU, t_{GPU}, includes the additional time of transferring data between CPU and GPU.

Finally, we conclude this section, with a proposition. For devices with compute capability 1.3, device memory accesses are scheduled on a per half-warp basis, where as full warp basis is available for compute capability 2.0. A half-warp is a group of 16 consecutive threads. Half-warp threads are generally executed together. Half-warps are aligned. For instance, threads 0–15 will be in the same half-warp, 16–31 will be in the same half-warp, etc. Similarly, in case of warp, threads 0–31 will be in the same warp, 32–63 will be in the same warp, etc. The partitioning of a block of thread into half-warps and full warps is done by mapping a thread index to a number. When the block dimensions are (D_x, D_y), the thread (x, y) is mapped to the number $x + yD_x$.

Proposition 2.5. Suppose that threads l_1 and l_2 map to the numbers n_1 and n_2, respectively. Then l_1 and l_2 are in the same half-warp if, and only if $\lfloor \frac{n_1}{16} \rfloor = \lfloor \frac{n_2}{16} \rfloor$. Also, l_1 and l_2 are in the same warp if, and only if $\lfloor \frac{n_1}{32} \rfloor = \lfloor \frac{n_2}{32} \rfloor$.

Proof. Let $l_1 = (x_1, y_1)$ and $l_2 = (x_2, y_2)$. Then, $n_1 = x_1 + y_1 D_x$ and $n_2 = x_2 + y_2 D_x$. Suppose l_1 and l_2 are in the same half-warp say k^{th} half-warp. Then, $16(k-1) \leq n_1, n_2 < 16k$. $\Rightarrow \lfloor \frac{n_1}{16} \rfloor = \lfloor \frac{n_2}{16} \rfloor = k - 1$. Conversely, suppose, $\lfloor \frac{n_1}{16} \rfloor = \lfloor \frac{n_2}{16} \rfloor = k'$. Then, $16k' \leq n_1, n_2 < 16(k'+1) \Rightarrow l_1$ and l_2 are in the same half-warp. Similar arguments work for full warp. □

3. GPU metrics for Matrix-Matrix Multiplication

In this section, we describe the GPU metrics for matrix-matrix multiplication. We explain the importance of three important GPU metrics which we defined already, namely, average bandwidth utilization, volume of data transfer and acceleration ratio. A study on GPU metrics gives a complete analysis on better GPU configurations among given GPUs. Suppose, we are provided with a collection of GPUs, using GPU metrics, we can choose the best GPUs using certain standard algorithms. Also, we can suggest the block partitioning for a given algorithm in order to obtain much better efficiency.

3.1. **Matrix-Matrix Multiplication.**

In this section, let us explain matrix-matrix multiplication in GPU.

```
void cpumultipy(float *u, float *w, float *x, int m)
{ for (int i = 0;i < m; i++)
     for (int j = 0;j < m; j++)
      {    float temp=0;
           for(int k = 0; k < m; k++)
             temp+= u[i*m+k]*w[k*m+j];
           x[i*m+j]=temp;
      }
}
```

Figure 1. Matrix multiplication in CPU

Figure 1 represents a classical multiplication of two $m \times m$ matrices U and W stored in row-major order in one-dimensional arrays u and w. The result matrix $X = U * W$ is returned by a one-dimensional array x in row-major order.

Remark 3.1. Matrix multiplication makes $O(m^3)$ accesses to u and w and $O(m^2)$ accesses to x.

Let us tile an $m \times m$ matrix using $n \times p$ tiles. Assume that $n \mid m$ and $p \mid m$. Tile index is given as (a, b), where $0 \leq a < m/n$, $0 \leq b < m/p$. A GPU code uses a block of threads to compute a tile (more accurately, the sub-matrix corresponding to a tile) of the result matrix X. To compute the entire matrix X, we use an $m/n \times m/p$ grid of thread blocks with thread block (a, b) to compute (a, b) tile of X. In CUDA, a thread may determine the coordinates of the block (a, b), that is part of using the variables $blockIdx.x$ and $blockIdy.y$. Hence, $(a, b) = (blockIdx.x, blockIdx.y)$.

There are many different possibilities to implement a same GPU code. These possibilities are due to the usage of registers, shared memory and number of X elements computed per thread, and so on. Different implementations result in different performance. Depends on the size of sub-matrix of X, our performance changes.

3.2. $p \times p$ **sub-matrix of** X **using shared memory.**

To improve the performance of matrix multiplication in GPU, we resort to a block matrix multiplication algorithm in which each of U, W, and X is partitioned into m^2/p^2 $p \times p$ sub-matrices U_{ij}, W_{ij} , and X_{ij} . We assume that $p|m$. The algorithm computes X using the equation:

$$X_{ij} = \sum_{0 \le k < m/p} U_{ik} W_{kj} \tag{3.1}$$

In the strategy of this subsection, a block of threads computes one $p \times p$ sub-matrix of X and each thread computes one element of this sub-matrix. So, we use $p \times p$ thread blocks with thread (dimX, dimY) computing element (dimY, dimX) of the sub-matrix (observe the difference in the convention to name threads and matrix elements). The thread block that is to compute X_{ij} executes the following algorithm.

Algorithm 1 Matrix multiplication using shared memory

```
for $k = 0 \rightarrow m/p$ do
    $u1 \leftarrow u[threadIdx.y][k], v1 \leftarrow w[k][threadIdx.x]$
    $temp \leftarrow u1 * w1$ This update step accesses shared memory but not device memory
    $tx \leftarrow 16 * blockIdx.y + threadIdx.y * n$
    $ty \leftarrow 16 * blockIdx.x + threadIdx.x$
    $x(tx + ty) \leftarrow temp$
end for
```

Figure 2 gives the kernel code for the case p. We verified experimentally that $p = 16$ gives best performance for half-warp, whereas $p = 32$ for full warp. We now obtain the device-memory access statistics for $p = 16$. Since each thread computes a single value of x, the number of half warps is $m^2/16$. In each iteration of the for loop, a half warp reads 64 bytes of u values from a single 128-byte segment using a 64-byte transaction and 64 bytes of w values using another 64-byte transaction. Since the for loop is iterated $m/16$ times, a half warp makes $m/8$ 64-byte transactions of u and w together. A half warp also makes a 64-byte write transaction on x. So, the total number of device-memory transactions made by this code is $m^3/128 + m^2/16$. Each of these is a 64-byte transaction with 100% utilization. The volume is $m^3/2 + 4m^2$ and β is 100%. This algorithm uses 11 registers per thread and 2092 bytes of shared memory; it achieves an occupancy of 1.0.

4. GPU metrics for Matrix-Vector Multiplication

In this section, we describe the GPU metrics for matrix-vector multiplication.

Figure 3 represents a classical matrix-vector multiplication of an $m \times m$ matrix A with a vector x stored in row-major order in one-dimensional arrays A and x. The resultant vector $y = A * x$ is returned by a one-dimensional array y in row-major order.

```
__global__ void gpumultipy1by4sub(float *u, float *w,
float *x, int m)
  {
__shared__ float u1[p][p], v1[p][p];
int m1 = m/16; int m2 = m*16;
int i1 = i2= (16*blockIdx.y+threadIdx.y)*m+threadIdx.x;
int j1 = 16*blockIdx.x+threadIdx.y*m+threadIdx.x;
float temp = 0;
for (int s = 0; s < m1; s++)
{
u1[threadIdx.y][threadIdx.x] = u[i1];
w1[threadIdx.y][threadIdx.x] = w[j1];
__syncthreads();
for (int k = 0; k < p; k++)
temp += u1[threadIdx.y][k]*w1[k][threadIdx.x];
__syncthreads();
i1 += 16;
j1 += m2;
}
x[i2+ 16*blockIdx.x ] = temp;
} }
```

FIGURE 2. p × p submatrix with shared memory

```
  void cpumultipy(float *A, float *x, float *y, int m)
  { for (int i = 0;i < m; i++)
     for (int j = 0;j < m; j++)
      y[j]+= A[i*m+j]*x[j];
  }
```

FIGURE 3. Matrix-vector multiplication in CPU

Matrix-vector multiplication is a particular case of matrix-matrix multiplication as a vector is nothing but an $n \times 1$ matrix. But, to improve the performance of the matrix-vector multiplication, we increase the computational load per thread to better mask-device memory with arithmetic operations. We employ each thread to compute a $p \times 1$ sub-matrix of y using shared memory. That is, each thread will compute p values of y. Again, by experimental verification, we found that $p = 16$ gives best performance. Since a thread will compute its assigned p values of y incrementally, we allocate a thread p registers $y[0:p]$ to store the incremental values computed so far. When computations are done, the thread will write its p computed values to device memory. Figure 4 gives a complete code to multiply $m \times m$ matrix with an $m \times 1$ vector using shared memory.

```
__global__ void matrivectormul(float *A, float *x,
float *y, int m, int p)
{
__shared__ float xs[p];

float temp = 0;
for (unsigned int j = 0; j < (m-1)/p+1; ++j)
{
  if(j*p + threadIdx.x < m)
     xs[threadIdx.x] = x[j*p + threaIdx.x];
  else
     xs[threadIdx.x] = 0;
    __syncthreads();

for (unsigned int k = 0; k < p; k++)
 if(blockIdx.x*p+threadIdx.x < m && j*p+k<m)
  temp += A[j*p+(blockIdx.x*p+threadIdx.x)*m+k] * xs[k];
  __syncthreads();
}

if(blockIdx.x*p+threaIdx.x < m)
  y[blockIdx.x*p+threadIdx.x] = temp;
__syncthreads();
}
```

FIGURE 4. p × 1 submatrix with shared memory

As per the device-memory statistics, each thread computes p entries of y. So, a half-warp computes $16*p$ entries of y. Hence, the number of half warps is $m^2/(16*p)$. In each iteration, the threads of a half-warp use 2 128-byte transactions to read the required A values. The total number of transactions on A is $m^2/(16*p)*2*m/(2*p) = m^3/(16*p^2)$. Also, average bandwidth utilization is 100%.

In general, an $m \times n$ matrix A and $n \times 1$ vector multiplication has total number of transactions as $m^2n/(16 * p^2)$. Since, in each iteration, a half warp makes $2 * p$ 64 byte transactions on x, the total number of transactions for x is n/p. Each of these transactions has 100% utilization. Also the number of device memory write transactions for y is n/p and each has 100% utilization, where each half warp makes p 64-byte device memory transactions to write out the n entries of y. Combining transactions of A, x and y, we get a total of $\frac{m^2n}{16p^2} + \frac{2n}{p}$ device transactions. The volume is given by $\frac{m^2n}{16p^2} * 128 + \frac{n}{p} * 64 * 2$.

In the similar fashion, one can compute the vector addition of two vectors of size n, which requires $3 * n/p$ memory transactions with 100% utilization and $3 * n/p * 64$ volume.

5. **Results**

In this section, we discuss the GPU metrics for various p, m and n values. Also, we deal with GPU metrics computation of BICGstab algorithm.

Table 1 gives GPU metrics for matrix-matrix multiplication of two $m \times m$ matrices, whereas Table 2 provides GPU metrics for matrix-vector multiplication for an $m \times m$ matrix with a vector of size m.

Table 1. GPU metrics for different p and m dimensions in matrix-matrix multiplication

(p,m)	Transactions	V	β	Run time(s) in GPU	Gflops	$\mathfrak{A}$
(p,m)	$\frac{2m^3}{p^2} + \frac{m^2}{p}$	$\frac{128m^3}{p^2} + \frac{64m^2}{p}$	100%	-	-	-
(16,m)	$\frac{m^3}{128} + \frac{m^2}{16}$	$\frac{m^3}{2} + 4m^2$	100%	-	-	-
(16,2048)	$\sim 6.7 \times 10^7$	$\sim 4.3 \times 10^9$	100%	0.06	373	546
(16,4096)	$\sim 5.4 \times 10^8$	$\sim 3.4 \times 10^{10}$	100%	0.45	377	772
(16,16384)	$\sim 3.4 \times 10^{10}$	$\sim 2.2 \times 10^{12}$	100%	25.25	378	965

Table 2. GPU metrics for different p and m dimensions in matrix-vector multiplication

(p,m)	Transactions	V	β	Run time(s) in GPU	Gflops	$\mathfrak{A}$
(p,m)	$\frac{m^3}{16p^2} + \frac{2m}{p}$	$\frac{8m^3}{16p^2} + \frac{128m}{p}$	100%	-	-	-
(16,2048)	$\sim 2.0 \times 10^6$	$\sim 1.6 \times 10^7$	100%	0.01	373	246
(16,4096)	$\sim 1.7 \times 10^7$	$\sim 1.3 \times 10^8$	100%	0.15	377	352
(16,16384)	$\sim 1.1 \times 10^9$	$\sim 8.6 \times 10^9$	100%	12.25	378	465

The BiCGStab algorithm depicted in Algorithm 2 contains several different operations involving matrix-vector multiplications and reductions, daxpy-like vector operations and scalar parameter updates. For each dot product, we require n/p transactions for both vectors and hence the number of transactions is $2n/p + 1$. Volume is $128n/p + 4$. Similarly, vector addition requires, $3n/p$ device memory transactions with $V = 192n/p$.

BICGstab algorithm has 3 matrix-vector multiplication, 5 dot products and 6 daxpy-like operations. We assume that the algorithm converges after k iterations. Then, we have $2k+1$ matrix-vector multiplication, $5k$ dot products and $5k+1$ daxpy-like operations. Hence the total number of transaction for an $m \times m$ matrix is given by $(\frac{m^3}{16p^2} + \frac{2m}{p}) \times (2k+1) + 5k \times (\frac{2m}{p} + 1) + (5k+1) \times \frac{3m}{p}$. That is, $\frac{(2k+1)m^3}{16p^2} + \frac{(29k+5)m}{p} + 5k$ and $V = \frac{(2k+1)*128m^3}{16p^2} + \frac{64*(29k+5)m}{p} + 20k$. Table 3 gives GPU metrics measured for BICGStab algorithm.

Algorithm 2 BiCGStab Algorithm

```
$s = AP$                                        Matrix vector Multiplication
$s = R - s$
$\hat{s} = s, u = v = q = 0, \alpha = \omega_0 = \rho_0 = 1$
while $(\sum s.s - \epsilon > 0)$ :             Scalar Product
$\rho = \sum s.\hat{s}$                         Scalar Product
$\beta = \rho/\rho_0 * \alpha/\omega_0$
$u = s + \beta(u - \omega_0 v)$
$v = Au$                                        Matrix vector Multiplication
$\alpha = \rho / \sum \hat{s}.v$                Scalar Product
$s = s - \alpha v$
$q = As$                                        Matrix vector Multiplication
$\omega = \frac{\sum q.s}{\sum q.q}$            Two Scalar Products
$P = P + \alpha u + \omega s$
$s = s - \omega q$
$\rho_0 = \rho, \omega_0 = \omega$
```

TABLE 3. GPU metrics for different p and m dimensions

(p,m)	Transactions	V	β	Time	Gflops	$\mathfrak{A}$
(p,m)	$\frac{(2k+1)m^3}{16p^2} + \frac{(29k+5)m}{p} + 5k$	$\frac{(2k+1)*128m^3}{16p^2} + \frac{64*(29k+5)m}{p} + 20k$	100%	-	-	-
(16,2048)	$\sim (k+\frac{1}{2}) * 4.1 \times 10^6$	$\sim (k+\frac{1}{2}) * 5.3 \times 10^8$	100%	10.06	373	376
(16,16384)	$\sim (k+\frac{1}{2}) * 2.1 \times 10^9$	$\sim (k+\frac{1}{2}) * 2.8 \times 10^{11}$	100%	45.25	378	689

The entire calculation in this paper has done in NVIDIA Tesla M2050, which has the following specifications.

Table 4. NVIDIA Tesla M2050 Specifications (Source: NVIDIA [4])

Number of Transistors	3.0 Billion
Number of GPUs	1
CUDA Cores	448
Streaming Multiprocessors	14
Core clock	1.15 GHz

6. Conclusions

Mathematical definition of GPU metrics helps us to find the optimum block size of a grid. Optimum block size suggests us to design our algorithm to divide a given matrix into blocks of matrices to gain the efficiency of GPU with 100% utilization. Further, one can store the matrix in sparse format and compute the number of transaction and volume for different sparse structures.

7. Glossary

$\lceil x \rceil$ - Ceiling function, $\min\{n \in \mathbb{Z} : n \geq x\}$
$\lfloor x \rfloor$ - Floor function, $\max\{n \in \mathbb{Z} : n \leq x\}$
η - number of registers
α - shared memory size
μ - number of threads
$\zeta_{\max}$ - maximum number of blocks per Streaming Multiprocessor (SM)
ϕ - maximum number of threads per warp
$\gamma_{\max}$ - maximum number of warps per SM
η_U - register unit
α_U - shared memory unit
γ_{AU} - warp allocation unit
γ_{SM} - number of warps co-scheduled in SM
ζ_{SM} - number of blocks co-scheduled in SM
γ_{PB} - number of warps per block
μ_{reqB} - number of threads required in a block
η_{reqT} - number of registers per thread
γ_G - number of registers assigned to block
α_{reqB} - number of bytes allocated to shared memory for a block
η_{reqW} - number of registers required in a warp
η_A - number of registers allocated to a warp
γ_{reqT} - number of register required per thread
ξ_{PB} - required memory per block
α_{PB} - required shared memory per block

η_{PB} - required registers per block

ψ - blocks co-scheduled per SM

$\mu_{B\max}$ - maximum number of threads in a block

$\eta_{T\max}$ - maximum number of registers per thread

$\zeta_{\lim}|_{\gamma}$ - limited blocks due to warps

$\zeta_{\lim}|_{\eta}$ - limited blocks due to registers

$\zeta_{\lim}|_{\alpha}$ - maximum blocks due to shared

REFERENCES

[1] NVIDIA CUDA Programming Guide, Version 4.2, 2012.Website: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] **Jason Sanders and Edward Kandort**, *CUDA by example, An Introduction to General-Purpose GPU Programming*, Addison-Wesley, 2011.

[3] CUDA Occupancy Calculator, NVIDIA Inc, Website: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[4] NVIDIA, Inc,Tesla M2050 and Telsa M2070 slot computing processor modules, Website: http://www.nvidia.com/docs/IO/43395/BD-05238-001_v03.pdf