A Performance Criteria for parallel Computation on basis of block size using CUDA Architecture

Ashis Kumar Dash

Abstract — GPU based on CUDA Architecture developed by NVIDIA is a high performance computing device. Multiplication of matrices of large order can be computed in few seconds using GPU based on CUDA Architecture. A modern GPU consists of 16 highly threaded streaming multiprocessors (SMs). GPU named Fermi consists of 32 SMs. These are computing intensive devices. GPUs have been found to be the best platform for massive data parallelism. CUDA architecture is based on the heterogeneous platform comprising of both CPU and GPU that offers enormous potential to solve complex harder problems with high speed. In most applications the sequential part of a program is executed using CPU and numeric intensive part on GPU. But mere execution of numeric intensive part on GPU will not increase the performance of the computation. Since GPU consists of highly threaded multiprocessors, threads must be well organised into Grids and Grid into blocks to maximize performance of parallel computation, depending upon architecture of the GPU. In this paper an organization of threads of a particular GPU is discussed and block size is determined to maximize the performance of parallel computation through matrix multiplication.

Keywords: CUDA, GPU, SM, Kernel, Grid, Block, threads, warps, matrix multiplication, parallel computation

1 INTRODUCTION

NVIDIA developed CUDA architecture is a platform for massive data parallelism.GPU performs the computation part in parallel using its highly threaded multi-processors. SMs produce threads which perform computation in parallel. This generation and assignment of threads is user defined. User has to write the programme to perform this. To write the program for GPU computation, one has to understand the GPU architecture well along with CUDA architecture. Mere program writing will not optimize the performance. User has to write the program based on architecture to optimize the time complexity. Therefore GPU generated threads are grouped into Grids, Grid into Blocks. Parallel computation of data is assigned to threads in block by block. Therefore Grid and block size play an important role to optimize the time complexity in parallel computation.

2 NVIDIA GPU Architecture

A modern GPU is organized into 16 highly threaded streaming multiprocessors (SMs). A pair of SMs forms a building block of a GPU. Each SM has 8 streaming processors (SPs). So a GPU consists of 128 SPs. Each SP has a multiply- add (MAD) unit, and an additional multiply unit, all running at 1.35 gigahertz. Newly developed GPU Fermi has 32 SMs. So Fermi consists of 256 SPs.

Architecture of a CUDA-capable GPU

3 CUDA Threads

The fundamental means of parallel execution in CUDA is fine-grained data parallel threads. Launching a CUDA kernel creates a grid of threads. The kernel function specifies the statements that are executed by each individual thread created when the kernel is launched at run-time.

3.1 CUDA Thread organization

Kernel function is a device function which is executed in GPU. Once kernel is invoked it generates grid of threads. All threads execute the same kernel function. These threads have unique coordinates to distinguish

themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy using unique coordinates, called blockId and threadId, assigned to them by the CUDA run time system. The blockId and threadId appear as built-in variables that are initialized by the run-time system and can be accessed within the kernel functions. When a thread executes the kernel function, references to the blockId and threaded variables return the appropriate values that form coordinates of the thread.

3.2 Thread assignment

CUDA run-time system generates the grid of threads once a kernel is launched. These threads are assigned to execution resources on a block by block basis. In the GeForce-8 series hardware, the execution resources are organized into Streaming Multiprocessors. For example, the GeForce 8800GTX implementation has 16 Streaming Multiprocessors. Up to 8 blocks can be assigned to each SM in the GeForce 8800GTX design as long as there are enough resources to satisfy the needs of all the blocks. In situations where there is an insufficient amount of any one or more types of resources needed for the simultaneous execution of 8 blocks, the CUDA run time automatically reduces the number of blocks assigned to each Streaming Multiprocessor until the resource usage is under the limit. With 16 Streaming Multiprocessors in a GeForc 8800 GTX processor, up to 128 blocks can be simultaneously assigned to Streaming Multiprocessors. Most grids contain much more than 128 blocks. The run-time system maintains a list of blocks that need to execute and assigns new blocks to Streaming Multiprocessors as they complete the execution of blocks previously assigned to them. In the GeForce 8800GTX design, up to 768 threads can be assigned to each SM. This could be in the form of 3 blocks of 256 threads each, 6 blocks of 128 threads each, etc. It should be obvious that 12 blocks of 64 threads each are not a viable option since each SM can only accommodate up to 8 blocks. With 16 SMs in GeForce 8800GTX, there can be up to 12,288 threads simultaneously residing in SMs for execution. So there is a limitation of assigning number of threads to a SM.

3.3 Thread scheduling

In GeForce 8800GTX once a block is assigned to a Streaming Multiprocessor, it is further divided into 32-thread units called Warps. The warps are implementation specific and can vary from one implementation to another. Warps are not part of the CUDA language definition. Warps are unit of thread scheduling. Knowledge of warp helps to optimize the performance of CUDA applications. Suppose a block has 256 threads. Then it has 256/32 =8 wraps. A SM has maximum 768 threads. That implies up to 24 warps can reside inside a SM at any point of time. For the GeForce-8 series processors, there can be up to 24 warps residing in each Streaming Multiprocessor at any point in time. The SMs are designed such that only one of these warps will be actually executed by the hardware at any point in time. A legitimate question is why we need to have so many warps in an SM considering the fact that it executes only one of them at any point of time. The answer is that this is how these processors efficiently execute long latency operations such as access to the global memory. When an instruction executed by threads in a warp needs to wait for the result of a previously initiated long-latency operation, the warp is placed into a waiting area. One of the other resident warps who are no longer waiting for results is selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution.

4 Problem Definition

Once kernel is lunched a grid of threads is generated. These threads are grouped into blocks. Parallel computation is performed by block of threads. Arbitrary block size will not improve the performance. To optimize the performance, block size should be well defined. But decision on block size is Architecture dependant.

5 CUDA Architecture based parallel matrix multiplication

Let M, N and P are three square matrices where M & N are input matrices and P is product matrix. The main steps in host code for matrix multiplication are illustrated below.

```
int main() {
1. // Allocate and initialize the matrices M, N, P
   // I/O to read the input matrices M and N
      ....
 2. // M * N on the device
    MatrixMulOnDevice(M, N, P, width);
 3. // I/O to write the output matrix P
    // Free matrices M, N, P
      ...
      return 0;
      }
```

The main program first allocates the M, N, and P matrices and then performs I/O to read M and N, in Part 1. Part 2 performs the matrix multiplication. After completing the

matrix multiplication in part 3, main function performs the I/O to write the product matrix P and free all the allocated matrices. The part 2 is the main focus. It calls a function, MatrixMulOnDevice() to perform matrix multiplication. The host code calls matrixMulOnDevice(), which is also executed on the host. It is responsible for allocating device memory, performing data transfers, and then activating the kernel that performs the actual matrix multiplication.

6 System Specification

For this report, the Machine that has been used has the following specification. It is a Hp xw8600 workstation. Its core is Intel Xeon E5405, core clock: 2000 MHz, FSB: 1333MHz, L2:12MB, Multiplier 6, socket LGA771,Data width:64 bit and its family is Harpertown. This machine possesses NVIDIA GPU, Quadro FX 3700. Its core is G92 with core clock: 500 MHz, Memory clock: 800 MHz, Memory size: 512 MIB, Memory type: 256- bit GDDR3, Memory bandwidth: 51.2, 112 number of streaming processors, active block capacity 768 and warp: 32 threads.

7 Performance of parallel matrix multiplication on basis of block size

The parallel matrix multiplication has been executed on the machine as described in subsection 6 using the algorithm section 5. The elements of the matrices are randomly generated floating point numbers of single precision. Here the variable block size has been taken to study the effect of block size. The execution time has been taken for matrix sizes, 1024x1024 and 1012x1012. This time, which is average of 100 readings, consists of time for transferring data to device and performing matrix multiplication on device. The aim of taking two different matrix sizes is to study the effect of block sizes those are divisors of matrix sizes. In this case 16 and 22 are divisor of 1024 and 1012 respectively. Here execution time is considered up to block size 22(22x22) as the maximum capacity of a block is 512 threads. A block size of 23(23x23) exceeds the number 512.Table-1 contains details of execution time.

TABLE-1

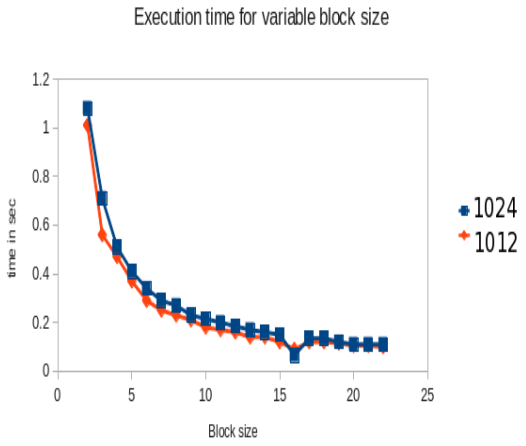
Block size	Execution time(sec) for Matrix Size (1024X1024)	Execution time(sec) for Matrix Size (1012X1012
2	1.08	1.01
3	0.71	0.56
4	0.51	0.47
5	0.41	0.37
6	0.34	0.29
7	0.29	0.25
8	0.27	0.23
9	0.23	0.21
10	0.215	0.18
11	0.20	0.17
12	0.185	0.16
13	0.17	0.14
14	0.16	0.14
15	0.15	0.12
16	0.065	0.095
17	0.135	0.12
18	0.135	0.12
19	0.12	0.115
20	0.11	0.105
21	0.11	0.105

Graph-1 below shows the variation of execution time. Blue and red curve are representing the execution time for matrix size 1024x1024 and 1012x1012 respectively. It decreases as block size increases from 2 to 16. It also decreases as block size increases from 17 to 22, but time for block size 17 to 22 are greater than the execution time of block size 16.

7.1 Observation

From graph 1 of the table 4, it is observed that block size of 16 has lest execution time. It is also observed that the execution time for block size 16 is not only lest but also its variation from execution time of block size 15 and 17 is significant. This is because of warp mechanism of CUDA architecture. So block size 16 is best choice for matrix multiplication on CUDA as long as the maximum capacity of a block is 512.

GRAPH-1



8 Limitation

This result is achieved from a particular system consists of particular GPU. This conclusion is not machine invariant. Using matrix multiplication as platform for parallel computation this conclusion is drawn. Data used for matrix multiplication is also specific (generated randomly up to some finite decimal places).

9 Conclusion

To optimize the performance of GPU, decision on block size is vital. This block size is machine dependant. Depending on architecture of GPU one has to decide the block size. Right block size for machine will optimize the performance.

10 Future Work

Above conclusion is architecture specific. No generalization of block size is done. Depending upon architecture, generalized optimum block size may be obtained which may be used for designing GPU architecture.

References

[1] NVIDIA CUDA Programming Guide. Version 3 http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_Program

[2] Alan Kaminsky, http://www.cs.rit.edu/ark/spring2009/736/ 4005-736 Parallel Computing II, GPU Computing: Introduction to GPGPU and CUDA.