Proceedings of the 2015 International Conference on Industrial Engineering and Operations Management
Dubai, United Arab Emirates (UAE), March 3 – 5, 2015

Optimizing SIMT Architecture using CUDA

Umahaeyo Christopher
Computer Engineering Department
Cyprus International University
Haspolat - Lefkosa, TRNC, Turkey
20122461@student.ciu.edu.tr

Oyku Akaydin
Computer Engineering Department
Cyprus International University
Haspolat – Lefkosa, TRNC, Turkey
oakaydin@ciu.edu.tr

***Abstract*— there exist a challenge as the true internal working of an algorithm depends on the method of exploitation, as well as hardware capability of the environment to which it is being exploited. In this paper, we perform parallel programming on matrix multiplication using CUDA on a GPU and make comparison between the results obtained from the sequential results on the CPU. It has been observed that, for large computations domains, the parallel implementation of the matrix multiplication provides a significant reductions.**

Keywords—GPGPU; CUDA; Parallel Computing; Topology.

I. INTRODUCTION

Today's Graphic Processing Unit (GPU) architecture supports different classes of applications with divergent classifications. Even though the industrial and scientific uses of the General Purpose Graphics Processing Unit (GPGPU) has evolved to the mainstream, it is evident however that there is general structure to all the applicative programs ported to the GPGPU. In computations that require large computations, the GPGPU delivers on demand amount of compute capability to the application. Examples are the real- time applications where satisfaction is needed for the complex on demand computation and deduction either in pixelated or mathematical form [1][2][7][8].

From the programmers perspective there exists a challenge as the true internal working of an algorithm and the method of exploitation, as well as hardware capability of the environment to which it is being exploited could be elusive. This paper exploits the use of an Nvidia GPU to which their parallel architecture are based on the SIMT architecture. The Single Instruction Multiple Threading (SIMT) architecture has similarities with the SIMD (Single Instruction Multiple Data) architecture. In section II below the SIMT architecture is discussed.

The mapping techniques that programmers use to increase performance when it comes to "threads", "warps", "blocks" and "grids" are varied from one algorithm to another. The model of programming for the Nvidia GPU used the hierchical parallelization of layer to which these are all present: grids, blocks, warps and threads. The final target in the hierarchy is the thread, which runs on the processor. It has become evident that understanding of this architecture has immense benefits in the optimum optimization of the GPU itself [7].

II. WHY SIMT?

In comparison to the SIMT, the SIMD architecture lacks the three key features enumerated below.

- Single Instruction, Multiple Registers
- Single Instruction, Multiple Addresses
- Single Instruction , Multiple Flow Path.

There exists a difference between the vector level parallelism of the CPU and GPU. While in the CPU the code compiler decides which segment is worthy of being parallelized, with respect to available instruction set, GPU however leaves this to the hardware to decide at runtime by both the scheduling and instruction dispatch unit [4][7].

A. Single Instruction, Multiple Registers

In the addition of two arrays, SIMD architecture breaks data into shorts parts while the loop processes them by ascribing short names. This is known as "short vector". In SIMD a loop is written for the executing of the instruction to enable concurrent data flow [3] [4].

```
__global__ void addKernel(int *C, int *A, int *B)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Fig 1: Example "Scalar Spelling" in SIMT architecture

On the other hand the keyword *__global__* instructs the GPU thread entry to *addKernel()*, which is called scalar spelling. *blockIdx*, *blockDim* and *threadIdx* are built in variable that helps to locate the exact placement of the thread's ID. This is seen in the Fig. 1 above. The *threadID*, however when executed isn't just a single number but a corresponding element index. In GeForce GT 440 used in this study, has 2 streaming multiprocessors (SMs). In each SMs there are 48 Shader Processor (SPs) and 8 Special Function Unit (SFU), with multiple cores present. Each core is saddled with running a thread. In comparison, SIMD executes data types with respect to its width and type while allocating register location to the data, but in SIMT the threads just *addKernel()* without wasting register bits and ALU circuitry. In cases where data is however inactive, registers are invoked to keep this data items while parallelism occurs.

B. *Single Instruction, Multiple Addresses.*

SIMT programmers benefit from rich syntax features which are more expressive with pointers being created carrying the value in all threads but might vary across threads as the instantaneous value is embedded in the *threadID*. When locating memory access either in DRAM, L2 cache, texture memory, constant memory or shared memory, the SIMT architecture allows for parallel consecutive threads access, but not consecutively. A directional random access mechanism. This particular feature allows for parallelization of SIMD programs that could not be paralleled. GPU hardware accesses DRAM in coalesce fashion of a single unit [5][9].

```
__global__ void addKernel(short *C, short *A, short *B)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    A[i] = B[C[i]];
}
```

Fig. 2: Indirect Memory Access in SIMT

During access to the shared memory, bank contention are resulted due to slow down random access. Even though theoretically mapping of banks and addresses will possibly reduce this, the hardware memory unit, could be tweaked to attend to only one access at a time. So shared memory is arranged into independent banks. For example, if the number of banks is *32* and *x* is a variable in shared memory then inferably, the traverse of the array, hits the bank with changes every 4 bytes. There exists hardware contentions when throughput peaks if all addresses are in all in different banks [5] [6].

C. *Single Instruction, Multiple Flow Path.*

Use of control instruction make direction of threads to several elements instead of one possible. In this a thread ID is issued equaling its index and step *nthreads*. From below it is noted that *if(A[i])* depends on the outcome of *A[i]*. The thread presented therein execute the code within the index, while some might be dropped. A divergent control flow of the threads. In comparison with SIMD, where store or load instructions are extended suppressing updates resulting in the vector register, SIMT remains clean and natural with no resulting extension for the vector register. During the CUDA program, when there is only one flow path, threads are in waiting when not running. It results into the shared of memory: fetch, decode and execute logic [3].

```
__global__ void find(int* A, int fEN, int* lEN, int* nTrack, int nthreads) {
  int tID = blockIdx.x * blockDim.x + threadIdx.x;
  int enWidth = 0;
  int* enList = lEN + tID*fEN;
  for(int i=tID; i<fEN; i+=nthreads) {
    if(A[i]) {
      enList[enWidth] = i;
      enWidth++;}
  } nTrack[tID] = enWidth;
}
```

Fig. 3: Flow Divergence in SIMT

In the "***if*** *stage"* the code is ran (full throughput - ***else*** threads wait -) and in the else stage (***if*** threads wait). This is known has divergence. To effectively parallelize this program, deeply nested control structures were written, which is however not recommended but actualized as sometimes divergence slows the randomization in accessing memory down. Indexing is however employed to avoid this contentions.

III. Harnessing The Architecture

GPU mode of computation of matrix multiplication is parallel compared to the CPU only which is serialized. In this study, we compare the results on CPU and GPU (Nvidia GeForce GT440). As mention earlier in Section II, there are 2 multiprocessor. This means for matrix multiplication, the maximum number of threads issued out of the matrix is 256 x 256 (65536 elements size). We however vary the elements size within the matrix from 1024 to 65536. In this study CUDA is used to exploit the SIMT architecture of the Nvidia GPU. The figure 4 illustrates the internal working on the kernel within the GeForce GT 440.

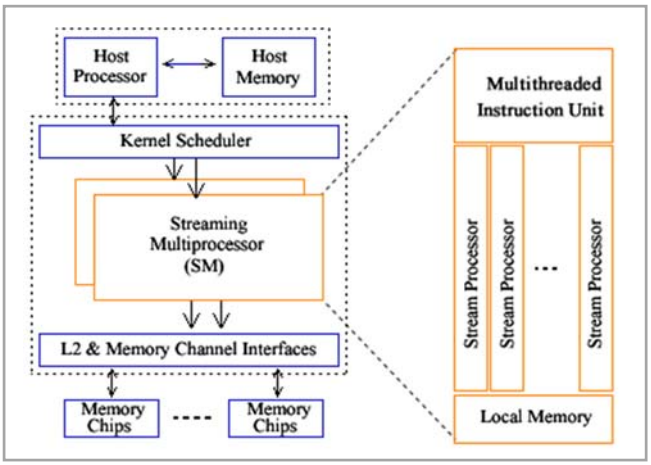

Fig. 4: Kernel Execution process Within GeForce GT 440 Architecture

When the matrix multiplication is launched, CUDA Kernel invocation is culled to kernel scheduler. With the CUDA kernel "grid and block" dimension specified, the actions of the scheduler becomes concurrent with the kernel call. The scheduler in-turn shares the work according to the resources available. It is in each of SMs that the number of grid and block specified and distributed accordingly.

In the SMs, the multithreaded instruction unit shares the issue a one time for all, instruction for all threads in the warp(s). All threads run the same code with each *threadID* using the compute memory and making control decisions. Since only one CUDA kernel launches a grid of threads blocks. The specification of this grid with respect to block and threads having in mind the elements size is what make topology of kernel execution tricky. The struck *dim3* is an integer vector type that helps in specifying the grid and block dimension for a typical kernel launch. For a 65536 elements sized matrix, in Fig. 6, there are 256 Grids with 1 thread in 256 blocks. While in Fig.7 there is 1 Grid with 256 threads in 1 block. Invariably there are 8 warps as a single warp can accommodate 32 threads. The variation in the dimension to

suit optimization of the program is further shown in Table I, below.

```
dim3 block(1, 1);
dim3 grid(WIDTH/block.x , HEIGHT/block.y);

multiply<<<grid,block>>>(Dev_A, Dev_B, Dev_C);
```

Fig. 5: Example Kernel Dimension on 65536 Element(s) Size

```
dim3 block(256, 256);
dim3 grid(WIDTH/block.x , HEIGHT/block.y);

multiply<<<grid,block>>>(Dev_A, Dev_B, Dev_C);
```

Fig. 6: Example Kernel Dimension on 65536 Element(s) Size

In Table I, the element size of the matrix multiplication was varied from 1024 to 65536. We varied the elements size on "CPU" only (sequential execution) launch and GPU (parallel execution) of the parallel program. In the heterogeneous execution, we varied the *dim3* struck for both grid and block dimension to observe the changes and results recorded to show performance with different grid/block dimension. Different GPUs has compute capability and might not fit with variation in the topology, as careful study in to the maximum thread per block, maximum threads per SM, wrap size, maximum grid and block dimension has to be done.

TABLE I: Execution Time For Sequential and Parallel of Matrix Multiplication With Varying Elements Size on GPGPU using CUDA.

Element Size	Sequential Exec. Time (µs)	Parallel Exec. Time (µs)
1024 (32 X 32)	0.0	0.07
4096 (64 X 64)	1.50	0.002
16384 (128 X 128)	10.90	0.002
65536 (256 X 256)	88.90	0.002

IV. Results and Discussion

As stated in III, the dimension to which the kernel is executed affects the length of time it takes for execution. For a particular elements size there is a corresponding kernel dimension that allows for minimum execution time.

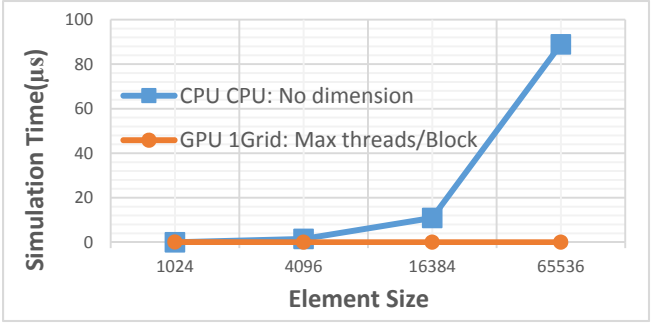

Fig. 8: Plot of Simulation Time for CPU and Varying Dimension on GPU against Element Size(s)

Illustrated in Figure 7 above is the simulations time (µs) against the varied element size. In above plot dimension of Max Grid means: that for a 1024 elements 32 Grid: 1 thread element in 32 blocks present in each Grid. Also at 4096 (64 x 64), there are 64 Grids, with 64 Blocks and 1 thread each in block and so on with respect of the element sizes. This form of kernel dimension is illustrated in Fig. 6, however means 1024 elements there exist the 1 grid with 32threads/block dimension is hence is optimized, as thread execution is not cued for execution as against the Fig. 5.

In 1 Grid with 256 threads in 1 block has a parallel simulation time at dim3 *block(256, 256)* for 65536 elements size to be 0.002µs, compared to sequential execution time on the CPU speedup actually took place as the return time is reduced. Please also refer to Table I.

In this paper speed up is calculated as:

$$S_{(N)} = T_{(CPU)} / T_{(GPU)} \quad (1)$$

$S_{(N)}$ represents speedup with respect to number threads. $T_{(H)}$ represents the time (µs) for sequential execution of the kernel on the CPU (Host Only code) and $T_{(K)}$ stands for time taken for parallel execution with respect to the varying element sizes.

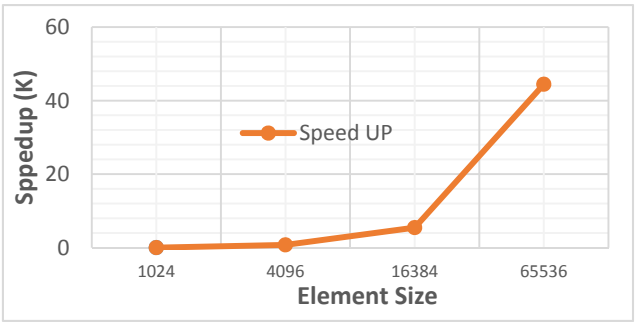

Fig. 8: Speedup against the Element(s) Size; with different Dimension

Figure 9 above illustrate the speedup realized when varying element sizes with difference dimension of kernel execution as seen in figures 6 and 7. As discussed above also, *Maximum Grid: 1thread/Block* (Figure 7) gives almost insignificant speedup while *1Grid: Maximum threads/Block* give visible speedup as elements size is varied with peak at 65536 element size.

V. Conclusion

When dimension as seen in Fig. 6 is used for a 65536 matrix elements size during multiplication, the kernel execution is least and speedup is realized with performance at its best. The effects of dimension of kernel call in a CUDA program conforming to a SIMT architecture is shown in Fig. 7 and Fig. 8. A programmer can therefore take charge of the architecture therein a typical Nvidia GPU to unlock its performance. This study also makes clear that parallelization of matrix multiplication produces significant speedup as computational needs increases.

VI. REFERENCES

[1] John D Owens, Mike Houston,David Luebke, Simon Green, John E. Stone and James C. Phillips. "GPU Computing". Proceeding of the IEEE | Vol. 96, No. 5, May 2008.

[2] Xiang Cui, Yifeng, and Hong Mei. "Improving Performance of Matrix Multiplication and fft on GPU". Key Laboratory of high confidence software Technologies, Ministry of Education, School of Electronics Engineering azazmany Peter Catholic University, Budapest, Hungarynd Computer Science, Peking University, Beijing, China. cuixiang@sei.pku.edu.cn,{cyf,meih}@pku.edu.cn

[3] Endre Laszlo z, Michael B. Giles , Jeremy Appleyard and Peter Szolgay, "Methods to Utilize SIMT and SIMD Instruction Level Parallelism in Tridiagonal Solvers", NVIDIA, Corporation, United Kingdom, mike.giles@maths.ox.ac.uk, flaszlo.endre,szolgayg@itk.ppke.hu

[4] Santonu Sarkar, Sayantan Mitra, Ashok Srinivasan, "Reuse and Refactoring of GPU Kernels to Design Complex Applications," Infosys Labs, Infosys Ltd. Bangalore 560100, India Dept. of Computer Science, Florida State University, 2012 10th IEEE International Symposium on Parallel and Distributed Processing with ApplicationsTallahassee, FL, asriniva@cs.fsu.edu 32306, USA, {santonu sarkar01,sayantan mitra01}@infosys.com

[5] David B. Kirk, Wen-mei W. Hwu, "Programming massively Parallel Processors, A hand-on Approach". 2010, Published by Elsevier Inc.

[6] Yunsup Lee, Avizienis R, Bishara A, Xia R, Lockhart D, C, Asanovic K, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators", Computer Architecture (ISCA), 2011 38th Annual International Symposium, Publication Year: 2011 , Page(s): 129-140

[7] Fan Wu, Miguel Cabral, Jessica Brazelton, " High Performance Matrix Multiplication on General Purpose Graphics Processing Units", Computer Science Department Tuskegee University Tuskegee, Alabama USA.

[8] Ping Guo, Liqiang Wang, Auto-Tuning, "CUDA Parameters for Sparse MatrixVector Multiplication on GPUs", Department of Computer Science University of Wyoming, USA, pguo@uwyo.edu

[9] Tekesha Athil, Richard Christian, and Yenumula B Reddy, "CUDA Memory Techniques for Matrix Multiplication on Quadro 4000," Department of Computer Science Grambling State University

VII. BIOGRAPHY

Umahaeyo Christopher is currently a masters degree student in Computer Engineering Department at Cyprus International Univesity, Haspolat - Lefkosa, TRNC, Turkey. Previously studied Electrical and Computer Engineering at Federal University of Technology in Minna, Niger State, Nigeria – Bachelor in Engineering - He is a student Member of IEEE Turkey.

Oyku Akaydin is an Assistant Professor in the Computer Engineering Department at Cyprus International University . She received her B.S., M.S. and Ph.D. degrees in Computer Engineering Department from Eastern Mediterranean University, Gazi-Magusa TRNC Turkey . Her research interests are parallel programming, parallel architectures and GPGPU computing.