Tranformation of CPU-based Applications To Leverage on Graphics Processors using CUDA

[1]Fawnizu Azmadi Hussin, [2]Anas Mohd Nazlee and [3]Noohul Basheer Zain Ali
Electrical and Electronics Engineering Department, Universiti Teknologi PETRONAS, 31750, Perak, Malaysia
[1]fawnizu@petronas.com.my, [2]verachens@gmail.com, [3]noohulbasheer_zainali@petronas.com.my

***Abstract*— Scientific computation requires a great amount of computing power especially in floating-point operation but a high-end multi-cores processor is currently limited in terms of floating point operation performance and parallelization. Recent technological advancement has made parallel computing technically and financially feasible using Compute Unified Device Architecture (CUDA) developed by NVIDIA. This research focuses on measuring the performance of CUDA and implementing CUDA for a scientific computation involving the process of porting the source code from CPU to GPU using direct integration technique. The ported source code is then optimized by managing the resources to achieve performance gain over CPU. Successful attempt at porting Serpent encryption algorithm and Lattice Boltzmann Method provided up to 7 times throughput performance gain and up to 10 times execution time performance gain respectively over the CPU. Direct integration guideline for porting the source code is then produced based on the two implementations.**

Index Term-- **parallel computing; GPU computing**

I. INTRODUCTION

Compute Unified Device Architecture (CUDA) is an architecture designed by NVIDIA Corporation for General Purpose Computing using Graphic Processor Unit (GPGPU), the term which become increasingly popular as the trend grows in year 2002 [1]. In the early stage of GPGPU, one of the popular known methods for computation is using OpenGL. This method requires the algorithm to be reconstructed to match graphic processes (i.e. using textures and such) and constrained to a lot of limitations such as access to memory and data type limitation to only floating-point.

CUDA-enabled graphic cards posses a higher computational performance (measured in FLOP/s) and higher bandwidth (measured in GB/s) when compared to CPU [2]. Although still to be further developed, early results suggested CUDA performance is indeed promising. The pinnacle of CUDA is demonstrated with the release of NVIDIA Tesla Personal Supercomputer which based on Tesla accelerator cards (similar with CUDA GPU and in fact it is the origin of CUDA development [2]) and also the latest development of "Tsubame" supercomputer using CUDA GPUs which was ranked 29th fastest in the world [3].

Currently, there are not many applications that use CUDA that could be acquired out of the box. Many applications would require the source code to be ported and compiled using CUDA compiler. The process of porting of source code could sometimes produce inefficient code that performs worse than the CPU counterpart or only little performance gain. General guidelines are needed for the process of porting the source code efficiently with considerable performance gain without much time spent for the process.

II. LITERATURE REVIEW

A. *Computing with CUDA*

Since CUDA has been primarily used for computation, it is very closely related to General-Purpose computing on Graphics Processing Unit (GPGPU). Owens *et al.* [4] describes the detail analysis of current GPGPU trend in architecture and also implementation. The architecture of CUDA GPU based on G80 architecture is reviewed to shed lights on multiprocessing and stream processing of current GPU. The paper [4] also explains the software performance libraries implementation (e.g. in the case of CUDA, CUBLAS and CUFFT libraries) and also the kernel performance which play a great role in CUDA performance.

For the program to be able to handle multiple data in parallel, the programmer has to make the program initialize the kernel which runs on multiple threads at any given time. The program model is similar to Single Program Multiple Data (SPMD) model, with added advantage of scalability [5]. Fig. 1 illustrates a program flow in kernel block initialization and scaling of the number of threads used according to the block specification. Each block can have a number of threads assigned to it in either one-dimension or up to three-dimension. Threads in the block will continue to run until the function for the threads to be synchronized is called. At that point, advanced threads will hold and wait for other threads to finish execution until the thread synchronization point.

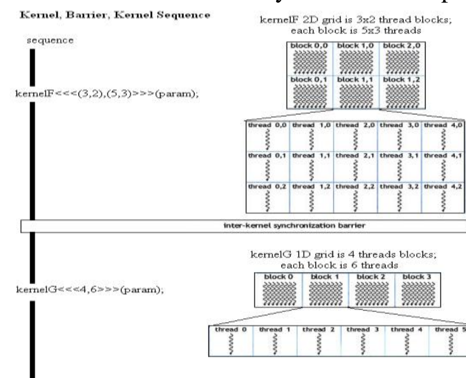

Fig. 1. Diagram of Single Program Multiple Data (SPDM) kernel

CUDA function is defined as a kernel that is called by specifying the number of thread blocks, the number of threads per block and the parameters needed for the function as shown in Fig. 1. Each thread executes the function in parallel and can communicate with other threads in the same thread block.

CUDA process flow is fairly straight-forward. Page-locked memory buffer is allocated in the host memory for faster memory transfer between host memory and device memory [2]. After the data transfer is done, the CPU will send instructions to the GPU for the program execution. The same program is executed on all of the threads inside the GPU. After

IJENS

all of the threads are synchronized, the data is transferred from the device memory back to the host memory. All of the process is summarized in Fig. 2.

As case studies, we considered the Serpent Encryption algorithm and the Lattice Boltzman algorithm for integer and floating point evaluations, respectively.

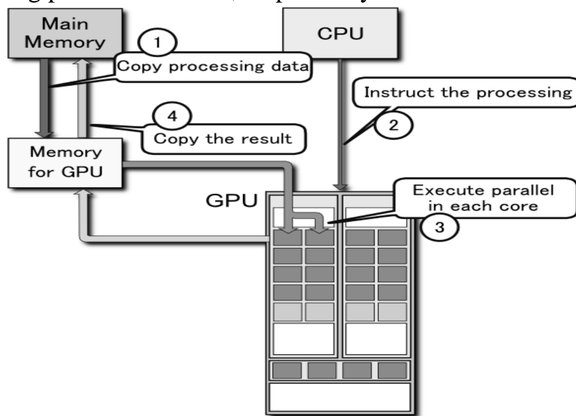

Fig. 2. CUDA process flow

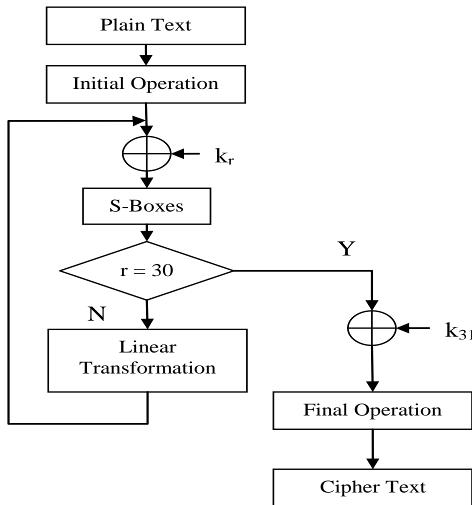

Fig. 3. Flow chart for Serpent Encryption algorithm

B. *Serpent Encryption Algorithm*

Serpent encryption operates based on 32-round SP-network with four 32-bit words as an input and up to 256-bit key as shown in Fig. 3. The design of Serpent algorithm is presented with parallelism by bit-slicing [6]. The encryption requires 132 32-bit words of key materials. The key length provided by the user is normally ranging from the minimum 128-bit to the maximum 256-bit key. Before any encryption could be done, the key provided by the user is expanded, mixed and went through S-boxes before becoming the key materials.

Graphic card as cryptography hardware is not entirely new, given the attempt is made around year 2005 using OpenGL for AES Cryptography [7]; however the performance suffered greatly from limited functionality. There were no successful attempts made after that until the arrival of CUDA. Manavski [7] managed to produce significant performance improvement using CUDA in AES Cryptography. The author managed to produce up to 20 times speedup over CPU using 8MB of data size and 128 bits of user key. Similar work is done for ARIA (cryptography originated from Korea), Yeom *et al.* [8] were able to produce comparable results to AES by effectively using shared memory and registers inside the GPU.

C. *Lattice Boltzmann for fluid flow*

Ever since GPU is introduced to render computer images, the floating-point operations (FLOP) performance has been increasing significantly till today that has already reached hundreds of GFLOP/s compared to CPU which focused mainly on the integer operation performance for mass consumer. By utilizing the GPU computation capabilities along with low CUDA learning curve, we would produce methodology for a simple porting process for additional performance gain.

Lattice Boltzmann has become increasingly popular in computational dynamics. Numerous attempts were made to increase the performance of algorithm that include study on the algorithm [9], optimization on multi-core platform [10] and multi-GPU implementation for Lattice Boltzmann [11]. All of the studies suggested significant performance gain by implementing the computation in parallel; however specific methodology for the implementation process on CUDA is not shown.

A phenomenon known as Karman Vortex Street is used for the computation fluid flow for this project. The unsteady separation of the flow of the fluid over a body causes a repeating pattern of swirling vortices that is illustrated in Fig. 4.

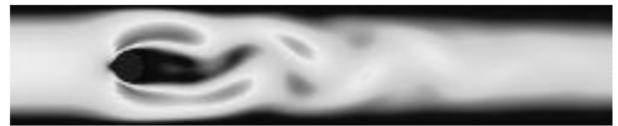

Fig. 4. Illustration for Karman Vortex Street phenomenon

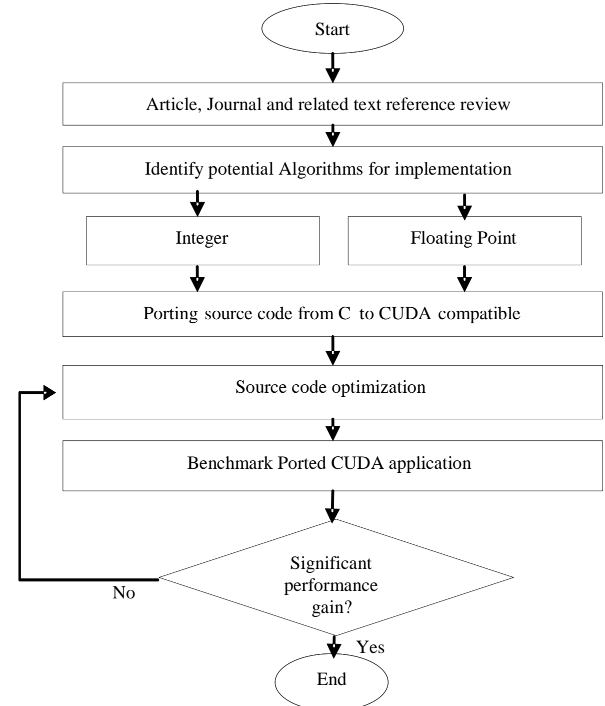

Fig. 5. Flow chart for process identification

III. METHODOLOGY

Fig. 5 shows the process identification flow chart.

A. *Potential algorithm for implementation*

Potential algorithm with source code written in C language could be considered a head start in porting the

algorithm. CUDA implements the extension of C language which can provide an advantage in porting the source code that is written in C language for the application, thus shorten the time needed for programming and more time for program debug and optimization. Direct integration of the source code is possible either by linking or rewriting portions of the source code to be executed on the graphic card. All of these will require knowledge of the selected application's algorithm to benefit from CUDA parallelism.

1) Integer operation

It is well known that graphic cards native operation is in floating-point for graphic processes. At a glance, integer operation might not receive any benefit from CUDA and the performance maybe less compared to CPU. This fact is supported by previous work by [12], as database operation mainly involves integer operation. Nvidia [2] also stated numerous clock cycles taken when performing integer operation. However, there are still integer operations that do not cost performance reduction that much, that is the bitwise operations. CUDA could sustain up to eight bitwise operations per clock cycle.

Cryptography is considered to have an intensive bitwise operations performed for converting plain text to cipher text. Recent success in implementing cryptography on CUDA for AES [7], DES [13] and ARIA [8] making it as motivation in using Serpent encryption as potential algorithm for the implementation. Additionally, Serpent encryption is distributed under GNU Public License (GPL), making it possible for modification and redistribution. A number of previous works in speeding up Serpent provided valuable data and information in providing basic understanding of Serpent performance so far.

2) Floating-point operation

Floating Point operation consisted of single precision and double precision. Each of them must correspond to IEEE-754 floating point standard for computation. Single precision computation performs faster compared to double precision however it is less accurate. CUDA support for double precision floating point is also limited to graphics card with *Compute Capability 1.3* [2]. Lattice Boltzmann computation utilizes double precision floating point and since the graphics card used supports the double precision features, it has become the motivation for the implementation.

B. Porting to CUDA and optimization

After the acquisition of the source code, the source code is divided into three sections that are data initialization, computation and result. Computation section is analyzed to identify the most compute intensive function. This can be done by using profiler or by manually adding in "Wall-clock" function to identify which computation took the most time. The compute intensive function is then converted as CUDA kernel function. This direct integration method is usually dubbed as naïve implementation method as some algorithms can already gain performance benefit using this method [13]. A function operating on an array containing a large number of elements can be used as an example.

1) Serpent encryption implementation

Data structures in the reference source code have been changed to allow parallel encryption to take place in each thread. Original data structures consist of one-dimensional arrays of four elements for the plain-text and cipher-text blocks (e.g. text[4]) while the key materials are stored in two-dimensional arrays of 33 x 4 elements (e.g. keys[33][4]). Another dimension would need to be added to the arrays to assign the thread number to the plain-text and key materials for each thread encryption. Instead of adding another dimension, we defined data structures that consist of four variables to accommodate the four 32-bit words plaintext and aligned them to 16-byte boundaries as shown in Fig. 6 for CUDA to read the data in a single instruction [2]. Similar structure is used for the keys, except the structure includes an array of 33 elements for the four variables. The defined data structures are used with pointers as dynamic data structures to take up variable number of blocks for encryption.

```
typedef struct __align__(16) {
     unsigned long x0, x1, x2, x3;
} SER_BLOCK;

typedef struct __align__(16) {
     unsigned long k0, k1, k2, k3;
} SUBKEY;

typedef struct {
     subkey k[33];
} SER_KEY;

typedef struct {
   uint32_t rkey[8];
} RAW_KEYS;
```

Fig. 6. Source code fragment for data structure

Modification was done to the encryption function by adding the thread number identification to the variable. This method could only be useful when using Electronic Codebook [14] as the Block Cipher Mode since no intermediate value is taken from previous or concurrent encryption. Partial source code of the parallel encryption is shown in Fig. 7. The variable "`idx`" in line 4 holds the thread number that is determined by calculation in which the data is encrypted. Each round of the encryption involves keying, passing through the S-Box and transformation as in lines 6, 7 and 8 respectively.

```
1 __global__ void cuda_encrypt(SER_BLOCK *enc_block,
2 SER_KEY *keys) {
3
4  int idx = (blockIdx.x * blockDim.x + threadIdx.x);
5
6 enc_block[idx] = keying(enc_block[idx],keys[idx].k[1]);
7 enc_block[idx] = SBOX00 (enc_block[idx]);
8 enc_block[idx] = transform(enc_block[idx]);
9   …
10 enc_block[idx] = SBOX31(enc_block[idx]);
11 enc_block[idx] = keying(enc_block[idx],keys[idx].k[32]);
11  }
```

Fig. 7. Source code fragment for parallel encryption in CUDA

The threads are managed through blocks that are identified by the "`blockIdx.x`" and "`blockDim.x`" in Fig. 7. CUDA limits the maximum number of threads to 512 for each block of threads. The resources used for the encryption is considered large, therefore the compiler has limited the number of threads to 256 threads per block. We have done performance study for the number of threads per block to study the effect of having various numbers of threads per block and discussed in Section IV.

The data size relates to the total number of threads used for the encryption. An example of 16KB of data size is shown in Fig. **8**. The number of threads used is the division of data size into 16-byte blocks as each encryption takes 16 bytes of input. The encryption also takes 32KB of user keys for the Key Scheduler to convert into key materials that are used for the keying processes in each round. In this implementation, the user keys are unique to each thread and do not share the same key.

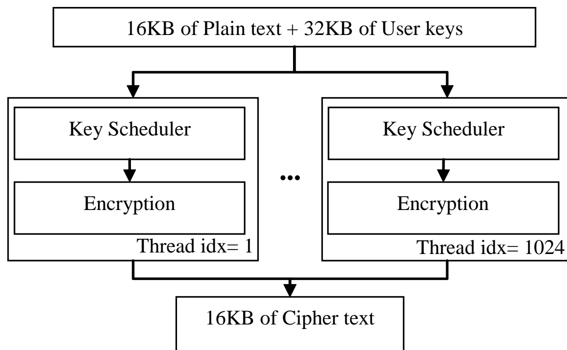

Fig. 8. Flow chart for multiple blocks encryption in parallel

2) Lattice Boltzmann for fluid flow

We used a simple simulation in utilizing the Lattice Boltzmann (LB) method to simplify the porting process. The program simulates Karman Vortex Street phenomenon that has been discussed in Section II of the report. The source code is acquired from an open source solution, that is the OpenLB [15] and it is still maintained by the users although depreciated. It is written in modular form for readability and flexibility for future extension. Unlike Serpent encryption, OpenLB consisted of many processes for a complete simulation. Thus porting the source code from C to CUDA-compatible would not be just one or two functions, but instead it needs several functions to perform correctly.

The collision simulation consisted of several modular functions that are differentiated according to the element position inside the lattice. The functions represent the computational dynamics for Bounce Back, Bhatnagar–Gross–Krook (BGK) for the bulk dynamics and all of the side boundaries include upper, lower, left and right. The elements function is determined in geometry initialization function.

We profiled the program by adding a “Wall-clock” function to the program for time measurement. The “Wall-clock” function measures the time taken from one point to another point in the program. Although, the timing might not be accurate compared to a real profiler application, it served the purpose in determining the ratio of a function runtime to the whole program execution time.

Based on the profile of the program, the function “Collide” occupies most of the program execution time. The goal of the initial attempt is to minimize the time taken for the function to execute and shorten the execution time altogether. Initial attempt was made by porting each of the function by using naïve implementation as described by the flow chart in Fig. **9**.

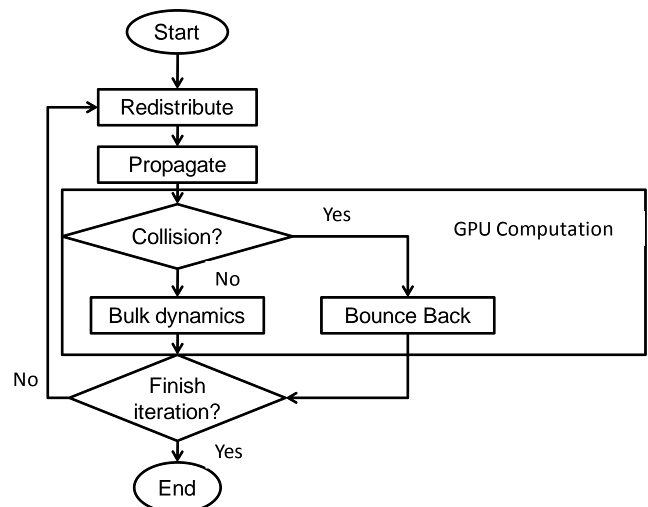

Fig. 9. Flow chart for initial implementation of LBM

Although this is the easiest method in getting the functions to be executed on the GPU, the code has become very inefficient. As a result, the performance for the implementation degraded significantly. Fig. **10** shows the comparison between CUDA implementation with CPU. The CPU compilers used as comparison are Intel C/C++ Compiler 11 (ICC) and GNU C Compiler 4.4 (GCC). The percentage time taken for the “Collide” function increased resulted in overall performance degradation.

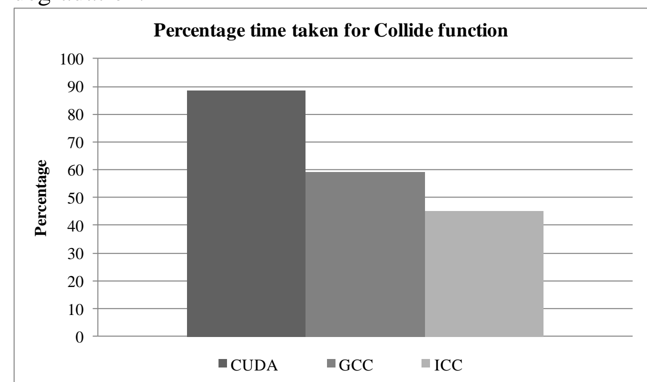

Fig. 10. Bar chart for the percentage time taken for function "Collide"

The performance degradation is caused by multiple requests for page-locked memory buffers. The time taken for the memory allocation occupies most of the function’s execution time. Although page-locked memory buffer offers high data transfer rate compared to usual memory allocation using standard C function, the amount of time taken for the transfer does not overcome the performance degradation caused by the page-locked request.

The complete application is made by including the data initialization and overall computation into the GPU. Since the initial value for the entire lattice elements are the same, the data initialization can be done in parallel using the GPU. Although the computation involved a few conditional branching, it is assumed the number of conditional branching is within the GPU limited capabilities. The general flow of the program is illustrated in Fig. **11**.

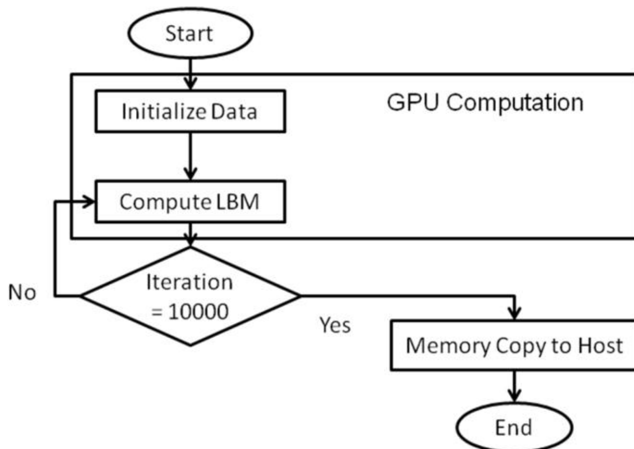

Fig. 11. Flow chart for complete application of LBM on CUDA

This is done not only to minimize the amount of size for memory transfer but also to reduce the complexity of the memory copy. Although CUDA supports multi-dimensional array operation, the array stored in the device memory is "flattened" (i.e. from to 2D array to 1D array) to reduce the complexity of the memory allocation. However, even though the array inside the GPU is flattened, it can still be addressed and computed in two-dimensional fashion. Fig. **12** illustrates the two-dimensional computing model supported by CUDA.

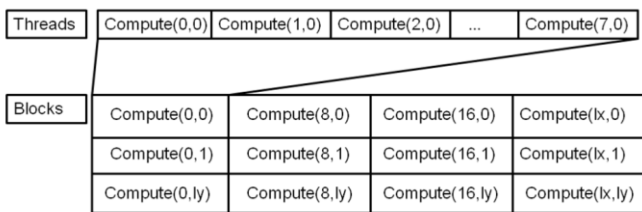

Fig. 12. Block diagram showing 2D computation model on CUDA

The array would require proper indexing for the 2D computation. For better resource management, only 1D thread with 2D blocks is used as the shared memory is limited to 16KB per block only. This provides acceptable range of number of threads per block that is from the minimum of 8 threads per block to 64 threads per block. Fig. **13** shows the indexing method used utilizing 1D thread and 2D blocks.

```
int idx = blockDim.x*blockIdx.x+threadIdx.x;
int by = blockIdx.y;
// node[x][y] = node_gpu[idx+lx*by];

node_gpu[idx+lx*by] =
compute(node_gpu[idx+lx*by];
```

Fig. 13. Source code fragment for indexing method

The direct integration for the complete application has shown significant performance gain in the execution time. The complete application was benchmarked with varying lattice size to study the performance effect. The result is further discussed in Section IV. After the successful source code porting process, an optimization was done to further improve the performance gain of the application.

The optimization done has affected only small part of the source code. Blocks and threads that are managed well will have equal distribution of resources and tasks, thus the device can perform efficiently. However, the higher the number of blocks or the number of threads per block does not mean it will give the better performance. In this case, we measure the performance for each number of threads per block to determine the best number of threads per block. Fig. **14** shows the result for the number of threads optimization with the result normalized over 8-thread per block.

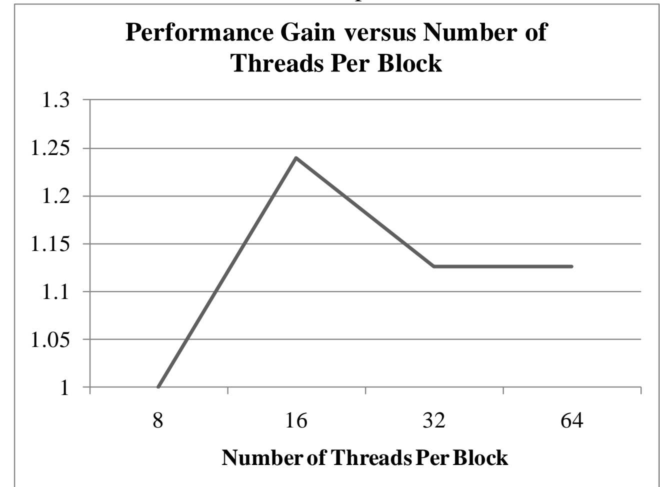

Fig. 14. Graph for number of threads per block optimization

IV. Results And Discussions

Based on the methodology presented, we have collected the performance results for our research to study the throughput performance by varying other factors such as data size for the input and the number of threads used. The test platform hardware is Intel Core 2 Duo Processor 2.2GHz and Nvidia Geforce GTX 260+ 896MB. GNU C Compiler 4.4 (GCC) and Intel C/C++ Compiler 11.0 (ICC) were used for the CPU while Nvidia CUDA Compiler 2.3 was used for the GPU. The optimized reference source code was used for CPU comparison. It was compiled with compiler flag "-O3 –march=core2" for GCC and "-O3 –xT –ipo" for ICC. CUDA source code was compiled with "-Xptxas –v –arch=sm_13" compiler flag. We have set a range of 5% accepted variation as the benchmark timing mostly in hundreds of millisecond and taken the average of three benchmark runs.

A. *Serpent Encryption on CUDA*

The throughput performance result is shown in Fig. **15**. The data size is varied to study the scalability of the CUDA encryption in handling different block sizes. The maximum data size achieved is 16MB, while at the same time still maintaining the throughput performance. As stated in the Section III, the number of threads used is scaled according to the data size; thus there is no trend visible in the throughput performance as seen in Fig. **15**. The amount of resources used to store the data is large and hardly fits into other types of memory, limiting the optimization that can be done through memory management.

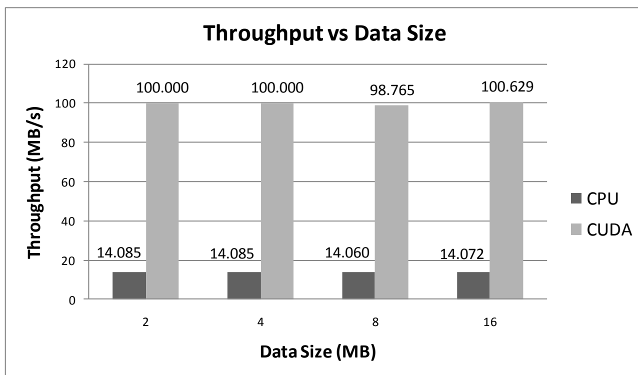

Fig. 15. Bar chart for throughput performance versus data size

The improvement achieved by minimizing the memory transfer is significant over previous implementation. The comparisons between complete application and initial attempt for the data size of 16MB are shown in Fig. 16. The throughput for CPU degraded in second attempt comparison as the benchmark also includes key scheduling process for CPU.

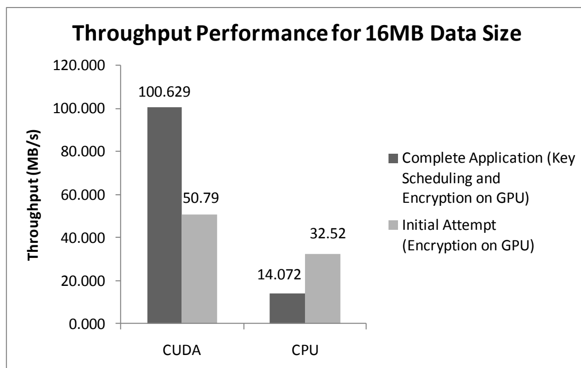

Fig. 16. Bar chart for throughput comparison between initial attempt with complete application

Although the total number of threads is determined by the data size, we must define the number of threads per block to have the application allocate the right number of blocks for the given number of threads per block. We varied the number of threads per block to study the effect on the performance throughput for further optimization.

The graph in Fig. **17** shows slight variation from 256 threads per block to 64 threads per block in the throughput performance. Performance degradation occurred as we further decrease the number of threads per block to 32 threads. The number of threads per block recommended by [2] is at least 16 threads and decreasing the number of threads any further will only result in performance degradation.

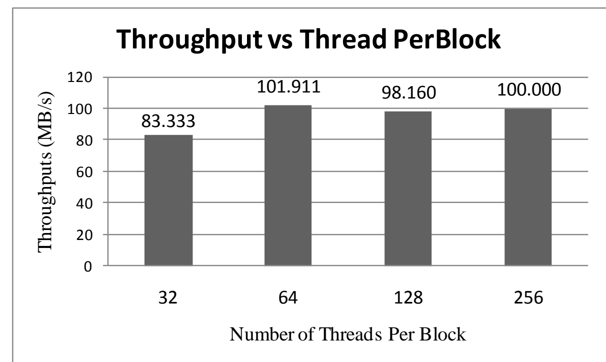

Fig. 17. Bar chart for throughput performance versus number of threads per block

B. *Lattice Boltzmann Method on CUDA*

Based on the direct integration implementation for Lattice Boltzmann method on CUDA, the performance of the implementation was measured against the CPU source code with GCC 4.4 compiler as reference and ICC 11.0 as high-performance compiler for CPU. The implementation managed a performance gain up to 10 times with the performance gain increase with the increment of the lattice size. The optimized thread number performance gain over the un-optimized thread number also increases as the lattice size increases. Fig. **18** shows the graph of the performance gain in terms of execution time with the result normalized over CPU based code compiled using GCC 4.4.

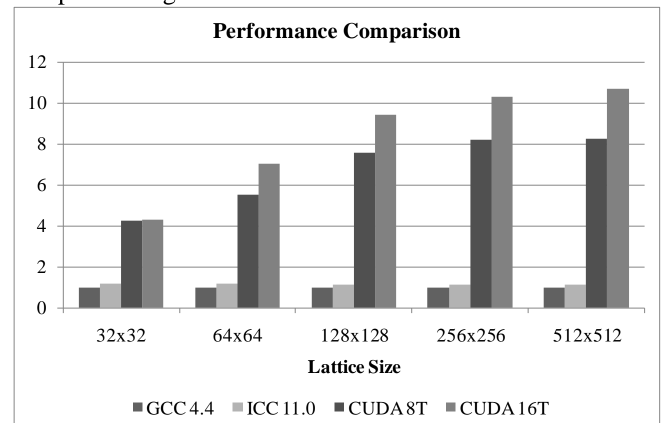

Fig. 18. Bar chart for performance comparison in execution time with varying lattice size

The data trend shows an increasing performance gain as the occupancy of the GPU increases and as the occupancy of the GPU nearing the maximum, the performance gain only increases slightly.

C. *Proposed General Guideline*

In short, the porting process of Serpent encryption algorithm and Lattice Boltzmann can be summarized into a number of key points:

- Identify parts in the source code for data initialization, compute intensive task and data retrieval.
- Aligning a structure type containing 16 bytes of data to 16-byte boundary provide coalesced memory transaction for the global memory thus increasing global memory bandwidth efficiency.
- Page-locked memory buffer can be requested for higher data transfer rate however multiple requests

over small amount of data degrade the overall performance significantly.

- Examine computation demanding task for data dependencies as well as sequence. This is important to recognize whether the computation can be done in parallel. Usually an arrays or matrices operation that is done in a loop without any dependencies can be directly ported.
- Although the global memory has the highest latency among the other types of memory in the device, the latency can be hidden by compute intensive instructions.
- Memory management is crucial in getting most of the performance from CUDA. By using shared memory as a temporary memory for operations could increase the performance.
- The amount of size for memory transfer must be kept at minimum as possible for efficiency. Generally, a big ratio between computations to memory transfer must be maintained.
- For multi-dimensional array computation, it is more convenient to flatten the array for memory allocation and memory transfer but the data will still be able to be computed in multi-dimensional computational model by managing the kernel's blocks and threads.

V. CONCLUSION

Based on the work done to both Serpent encryption and Lattice Boltzmann, a general guideline is produced for simple and direct implementation from CPU-based algorithm in C language to CUDA source code that can be executed on the GPU. Both of the programs are based on compute intensive application that usually stresses the CPU extensively.

Serpent encryption algorithm which is based on integer and bit-wise manipulation managed to achieve up to 7 times performance gain compared to a single-core CPU by using multiple block encryption in parallel. The data for the multiple blocks is handled independently without any relation with the other blocks. The Lattice Boltzmann was chosen for the floating-point implementation because of its parallel algorithm for the lattice's elements computation. By using direct integration method, the performance gain in the execution time is up to 10 times over that of the CPU.

Although CUDA is able to provide massive parallelism, it is limited by only one kernel can be executed at the same instance. Nevertheless, it was able to give considerable performance gain and still holds much potential. Direct integration method also worked well, given the programmer could spend some time in profiling the ported source code. Optimizations such as limiting memory transfer, thread block management and memory management could already give considerable performance gain over system with just CPU as computation processor.

REFERENCES

[1] M. Harris, "General-Purpose computation on Graphic Processing Units", [Online]. Available: http://www.gpgpu.org. Last Accessed: September 10, 2009.

[2] NVIDIA Corporation, "Programming Guide," in NVIDIA CUDA Compute Unified Device Architecture, Ver. 2, June 2008, [Online]. Available: http://www.nvidia.com/object/cuda_home.

[3] D. Adams, Inside Tsubame: Japan's NVIDIA GPU Supercomputer, Dec. 2008, [Online]. Available: http://www.osnews.com/story/20635/Inside_Tsubame_Japan_s_NVIDIA_GPU_Supercomputer. Last Accessed: May 3, 2009.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," in Proceedings Of The IEEE, Vol. 96, No. 5, pp 879-899, May 2008.

[5] J. Nickolls, I. Buck, M Garlanda and K. Skadron, "Scalable Parallel Programming with CUDA," in GPUs for Computing, Vol. 6, No. 2, April 2008.

[6] R. Anderson, E. Biham, and L. Knudsen, "Serpent: a proposal for the advance encryption standard." [Online]. Available: http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf, Accessed: July 1, 2009.

[7] S. A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in IEEE International Conference on Signal Processing and Communication, ICSPC 2007, Nov. 2007, pp. 65–68.

[8] Y. Yeom, Y. Cho, and M. Yung, "High-Speed Implementations of Block Cipher ARIA Using Graphics Processing Units," in Proceedings of the 2008 international Conference on Multimedia and Ubiquitous Engineering (April 24 - 26, 2008). MUE. IEEE Computer Society, Washington, DC, 271-275. 2008.

[9] Weibin Guo, Cheqing Jin and Jianhua Li, "High Performance Lattice Boltzmann Algorithms for Fluid Flows," Information Science and Engieering, 2008. ISISE '08. International Symposium on, vol.1, no., pp.33-37, 20-22 Dec. 2008.

[10] Williams, S., Carter, J., Oliker, L., Shalf, J. and Yelick, K., "Lattice Boltzmann simulation optimization on leading multicore platforms," Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on , vol., no., pp.1-14, 14-18 April 2008.

[11] Jifu Zhou, Chengwen Zhong, Jianfei Xie and Shiqun Yin, "Multiple-GPUs Algorithm for Lattice Boltzmann Method," Information Science and Engieering, 2008. ISISE '08. International Symposium on , vol.2, no., pp.793-796, 20-22 Dec. 2008.

[12] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in Proc. of SIGMOD, 2004.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. "A Performance Study of General Purpose Applications on Graphics Processors using CUDA," in Journal of Parallel and Distributed Computing, Elsevier, 68(10):1370-80, Oct. 2008, DOI http://dx.doi.org/10.1016/j.jpdc.2008.05.014.

[14] Dworkin, M. 2001. "Recommendation for block cipher modes of operation: methods and techniques." NIST Special Publication 800-38A.

[15] OpenLB (Open source lattice Boltzmann method), [Online]. Available: http://www.lbmethod.org/openlb/index.html. Last Accessed: October 19, 2009.