A Restructuring Algorithm for CUDA

M. A. Al-Mouhamed and A. H. Khan
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
{mayez, ahkhan}@kfupm.edu.sa

***Abstract*—Graphic processing Units (GPUs) are gaining ground in high-performance computing. CUDA (an extension to C) is most widely used parallel programming framework for general purpose GPU computations. However, the task of writing optimized CUDA program is complex even for experts. We present a method for restructuring loops into an optimized CUDA kernels based on a 3-step algorithm which are loop tiling, coalesced memory access, and maximizing machine utilization. For this we identify the GPU constraints for maximum performance such that the memory usage (global memory and shared memory), number of blocks, and number of threads per block. In addition we identify the condition for maximizing utilization of the GPU resources. We also establish the relationships between the influencing parameters and propose a method for finding possible tiling solutions with coalesced memory access that best meets the identified constraints. We also present a simplified algorithm for restructuring loops and rewrite them as an efficient CUDA Kernel. The execution model of synthesized kernel consists of uniformly distributing the kernel threads to keep all cores busy while transferring a tailored data locality which is accessed using coalesced pattern to amortize the long latency of the secondary memory. In the evaluation, we implement some simple applications using the proposed restructuring strategy and evaluate the performance in terms of execution time and GPU throughput.**

Keywords: CUDA, GPU, Parallel Programming, Compiler Transformations, directive-based language, source-to-source compiler, GPGPU

I. INTRODUCTION

Massively Parallel computing has obtained prominence through advances in implementing massive multithreading and recent improvements in its programming [1, 2, 3]. Recent development in Graphic Processing Units (GPUs) has opened a new challenge in harnessing their computing power as a new general purpose computing paradigm. Strong implications are expected on computational science and engineering, especially in the area of discrete numerical simulation [4].

Modern GPUs use multiple streaming multiprocessors (SMs) with potentially hundreds of cores, fast context switching, and high memory bandwidth to tolerate ever-increasing latencies to main memory by overlapping long-latency loads in stalled threads with useful computation in other threads [5]. The Compute Unified Device Architecture (CUDA) is a simple C-like interface proposed for programming NVIDIA GPUs. However, porting applications to CUDA remains a challenge to average programmers. CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer between the host and GPU memories, and of manually optimizing the utilization of the GPU memory [3].

Performance study of general-purpose GPU programming have been reported [6] for applications such as SRAD structured grid, back-propagation unstructured grid, data encryptions standard, Needleman – Wunsch dynamic programming, and k-means data mining. Impressive speedups ranging from 2.9 to 35 for the above applications have been achieved over single threaded programs. Some limitations have also been reported when the available parallelism is semi-static. A CUDA implementation for the gravitational N-body simulations using GPU is reported [7]. The GPU performs force calculation and updating, while the host CPU performs the predictor, corrector, and integration steps. Implementation is based on two direct N-body integration codes, using the 4th order predictor-corrector Hermite integrator with block time-steps, and one Barnes-Hut tree-code, which uses a second order leapfrog integration. The above implementation merely maps the computation of pairwise particle interactions onto the GPU which makes the time-consuming updating of the neighbor lists on the CPU a bottleneck since synchronization and frequent data transfer between CPU and GPU.

CUDA programming requires an expert level understanding of the memory hierarchy and execution model to reach peak performance. Even for experts, rewriting a program to exploit the architecture in achieving high speedup can be tedious and error prone. Several high-level interfaces [1, 2, 3] has been proposed to perform source-to-source translation based on programmer defined pragmas or annotations to generate CUDA programs with less burden to the programmers. Most execution of a scientific program is spent on loops. Compiler analysis and compiler optimizations have been proposed to make the execution of loops faster. CUDA-lite [1] is an experimental enhancement to CUDA that allows programmers to deal only with global memory with transformations to leverage the complex memory hierarchy. A set of annotations describing certain properties of the data structures and code regions designated for GPU execution are proposed. The tool analyze the code along with these annotations and determine if the memory bandwidth can be conserved and latency can be reduced by utilizing any special memory types and/or by massaging memory access patterns. Upon detection of an opportunity, CUDA-lite performs the transformations and code insertions needed. Authors claim the tool produces code with performance comparable to hand-coded versions.

A framework for source-to-source translation of standard OpenMP applications into CUDA-based code is proposed [2]. It has two phases: (1) a compile-time optimization techniques which applied parallel loop-swap and loop-collapsing, and (2) an OpenMP to GPGPU translation system. In the later step, partitioning and data mapping are used to convert work-sharing OpnMp constructs into kernel with defaults block size and number of blocks. Shared data are mapped to global memory. Thread private data are replicated and allocated on global memory for each thread. Private data are mapped to register banks assigned for each thread. Evaluation uses Jacobi, and SPMUL, and two NAS OpenMP Parallel Benchmarks (EP and CG). It is reported a performance improvements of up to 50x over the un-optimized translation (up to 328x over serial on a CPU.

A high-level directive-based compiler (hiCUDA) [3] is proposed to ease the task of writing CUDA programs. The compiler translates a hiCUDA program to a CUDA program using a computation model and a data model in which programmers allocate and de-allocate memory on the GPU and move data between the host memory and the GPU memory. Evaluation of five CUDA benchmarks (MM, CP, SAD, TPACF, RPES) shows that the provided simplicity and flexibility come at no expense to performance as execution times is within 2% of that of the hand-written CUDA version.

A source-to-source compiler transformation (CUDA-CHiLL) [8] aims at alleviating the need for understanding memory hierarchy and execution model in writing optimized CUDA programs. It proposes a source-to-source transformation based on the polyhedral program transformation and ChiLL framework which is capable of composing transformations while preserving the correctness of the program at each step. The authors claims that optimizing the BLAS library routines yields results comparable to hand-tuned versions in some cases and outperforming hand-tuned in other cases.

In this paper we present a method for restructuring loops into an optimized CUDA kernels based on a 3-step algorithm which are loop tiling, coalesced memory access, and maximizing machine utilization. For this we identify the GPU constraints for maximum performance such that the memory usage (global memory and shared memory), number of blocks, and number of threads per block. In addition we identify the condition for maximizing utilization of the GPU resources. We also establish the relationships between the influencing parameters and propose a method for finding possible tiling solutions with coalesced memory access that best meets the identified constraints. We also present a simplified algorithm for restructuring loops and rewrite them as an efficient CUDA Kernel. The execution model of synthesized kernel consists of uniformly distributing the kernel threads to keep all cores busy while transferring a tailored data locality which is accessed using coalesced pattern to amortize the long latency of the secondary memory. In the evaluation, we implement some simple applications using the proposed restructuring strategy and evaluate the performance in terms of execution time and GPU throughput.

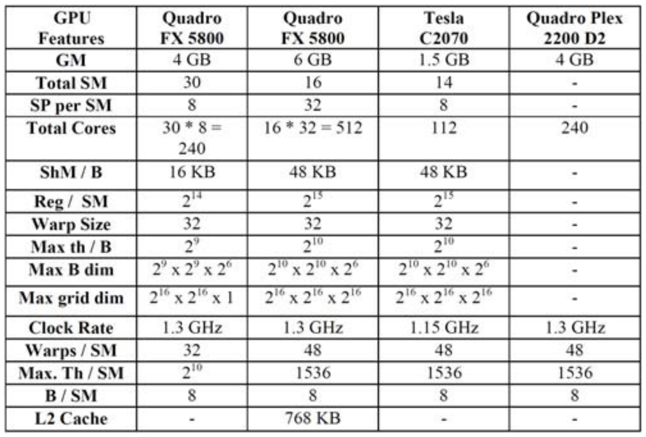
This paper is organized as follows. Section 2 presents some analysis of GPU that is critical for performance tuning. Section 3 presents a proposed approach for restructuring algorithm for CUDA. Section 4 presents the evaluation of applications and comments on execution times and throughput. Finally, Section 5 concludes about this work.

II. Background

Ideal GPU applications have large data sets, high parallelism (data parallelism), and minimal dependency between data elements [9].

A. GPU Architecture

It is organized into an array of highly threaded Streaming Multiprocessors (SMs). Each SM has a number of Streaming Processors (SPs) that share control logic and instruction cache. The total number of SMs in a device and SPs per SM can vary from one generation of GPUs to another generation. Each GPU currently comes with up to 4 GB of graphics double data rate (GDDR) DRAM referred to as global memory. These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold video images, and texture information for three-dimensional (3D) rendering, but for computing they function as very-high-bandwidth, off-chip memory, though with somewhat more latency than typical system memory. A general-purpose random access, readable and writable off-chip Global Memory (GM) visible to all threads in all blocks is provided. GM is the slowest of the available memory spaces, requiring hundreds of cycles, and is not cached. Each SP has a shared memory (ShM) which is on-chip, readable and writable, and visible to all threads running within SM and as fast as register access. However, ShM is very small in size compared to GM. Table 1 shows some published features of some popular GPUs.



GPU Features	Quadro FX 5800	Quadro FX 5800	Tesla C2070	Quadro Plex 2200 D2
GM	4 GB	6 GB	1.5 GB	4 GB
Total SM	30	16	14	-
SP per SM	8	32	8	-
Total Cores	30 * 8 = 240	16 * 32 = 512	112	240
ShM / B	16 KB	48 KB	48 KB	-
Reg / SM	2^{14}	2^{15}	2^{15}	-
Warp Size	32	32	32	-
Max th / B	2^{9}	2^{10}	2^{10}	-
Max B dim	$2^{9} \times 2^{9} \times 2^{6}$	$2^{10} \times 2^{10} \times 2^{6}$	$2^{10} \times 2^{10} \times 2^{6}$	-
Max grid dim	$2^{16} \times 2^{16} \times 1$	$2^{16} \times 2^{16} \times 2^{16}$	$2^{16} \times 2^{16} \times 2^{16}$	-
Clock Rate	1.3 GHz	1.3 GHz	1.15 GHz	1.3 GHz
Warps / SM	32	48	48	48
Max. Th / SM	2^{10}	1536	1536	1536
B / SM	8	8	8	8
L2 Cache	-	768 KB	-	-

Table 1: Some features for some NVIDEA GPUs.

GM is linked to the GPU device through a very large data path of 512-bits wide. Through such a bus width, sixteen consecutive 32-bits (4 bytes) words can be fetched from global memory in a single cycle. The on-chip memory resource includes register files (16K per SM, see Table 1), shared memory (16KB or more per SM). To hide the long off-chip memory access latency, a high number of threads are supported to run concurrently. The threads are grouped in blocks which will be scheduled to SMs dynamically on the availability of each SM. These threads follow the single-program multiple-data (SPMD) program execution model. Within a block, threads are grouped in 32-thread instructions called warps, where each warp is being executed in the single-instruction multiple-data (SIMD) manner within some SP.

B. CUDA Execution Model

A CUDA program is a unified source code encompassing both the host and the device code. It consists of one or more phases that are executed on either the host (CPU) or a device that is a GPU. The phases that exhibit rich amount of data parallelism are implemented in the device code. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures [10].

Figure 1 shows the execution hierarchy of a typical CUDA kernel function on a device. Each kernel initiates a set of blocks defined by the programmer as grid dimension with number of threads to be executed within each block while invoking the device kernel function. Now, the block scheduler dynamically schedules each thread block to one SM based on the availability of resources within SM [1]. An SM can handle at most 8 blocks at a time as we have 8 SPs within each SM (see Table 1). Also, the possible number of concurrent blocks per SM depends on the number of warps per block, number of registers per block, and the shared memory usage per block. These constraints will be developed in Section IV. An SM has a limited number of resources in terms of warps scheduling, registers, and the shared memory which are to be partitioned among the scheduled blocks in each SM. For many GPUs (Table 1), each SM can handle 32 warps at a time. Each SM has 32 KB registers and 16 KB of shared memory.

SM manages threads ids and threads execution. Threads within a block cooperate within SM using ShM while threads in different blocks cannot cooperate, not even using GM since the blocks are scheduled to different SMs dynamically by the scheduler. The data transfer between different blocks can be done by separate invocation of the kernel which will be serialized. So, in case of recurrence in application space, the whole recurrence must be contained in each single thread because serialization is controlled only by defining different kernels. Thus in the case of a recurrence, we may end up with a few very coarse threads, a situation that might lead to low GPU utilization which is discussed in details in Section IV.

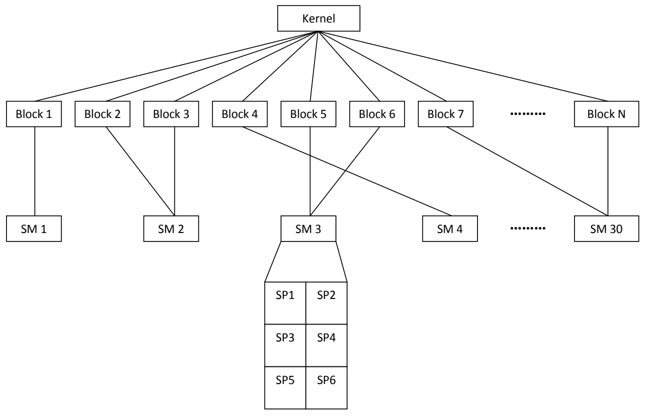


Figure 1: CUDA Kernel Execution Hierarchy

Each SM schedules one warp at a time with zero overhead warp and thread scheduling (using Scoreboard as a warp instruction scheduling). The warp is the unit of thread scheduling in SMs. Each warp consists of 32 threads of consecutive thread ids. In the case of higher dimensional kernels, warps will be retrieved from blocks according to the row major numbering. As warps executes in SIMD fashion, if there is a high latency exception such as loading data from GM or storing results to GM then the whole warp must be suspended and its context if preserved.

The major bottleneck to achieving performance while using CUDA is the memory bandwidth and latency. The GPU provides several different memories with different behaviors and performance that can be leveraged to improve memory performance. The transfer of data between GM and ShM is explicitly programmed by the user. A DMA operation is initiated by the SM whenever it finds one or more threads within a warp to perform such a long latency memory transfer operations (accessing global memory) and schedule another warp (ready to execute) to the SP [10]. This mechanism of filling the latency of expensive operations with work from other threads is often referred to as latency hiding.

GM is partitioned into segments of size equal to 32, 64 or 128 bytes (see Table 1) and aligned to this size. The elements in one segment can be accessed by a single memory transaction. By considering the largest segment size of 128 bytes and also the data path of 512 bits, the compiler issues a single load/store instruction for 16 consecutive elements accessed by 16-threads (half warp) to reduce the number of memory transactions of global memory. So, the performance of memory transfers can incredibly be improved through the use of coalesced global memory accesses that is accessing a regular pattern of consecutive elements by a half warp (16 threads) based on some conditions [1]. Therefore, if SPs are kept busy executing through warp switching then the whole transfer between GM and ShM is hidden by some execution which implies that the parallel program time does not account for such an expensive memory transfer. Since, shared memory is very small in size so we have to perform some loop transformation such as loop tiling, a mechanism to adjust loop execution to match with underlying machine or memory system, to make the availability of enough data for the warp per SP.

III. A Restructuring Algorithm for CUDA

In this section we proposed a CUDA kernel restructuring algorithm, a general strategy to achieve maximum possible performance by better utilization of the machine. In CUDA, the worker threads are identified by thread ID and being organized by blocks which are identified by block ID. This identification is used in a kernel to define a mapping of computations to threads (workers). An array of any dimension is accessed as a linear memory which is allocated in a row-major order. The objective of having multi-dimensional blocks of threads is to ease the mapping of computation results to the worker threads.

The proposed restructuring algorithm aimed at generating efficient CUDA kernels. It is based on following guidelines:

1. Tiling the code so that the aggregate data locality of a tile (block of threads) is fetched, and being small enough to fit, onto ShM prior to computations instead of direct load form GM, no matter whether using coalesced access or non-coalesced access.
2. Exploring different ways of mapping computations to threads to favor coalesced global memory access while loading from and/or storing into GM.
3. Increase thread granularity to amortize the ratio of data transfer per computation without having some SM being idle, i.e. low utilization of the available SMs and the SPs within each SM.
4. Reduce (1) the number of local variables (register use) and (2) block size, to avoid reducing the number of blocks that can handled by SM at a time which may affect overall GPU utilization.
5. Use kernel block size greater than or equal to tile size such that each thread in a block loads one element of a tile into ShM. This reduces instruction fetch and processing overhead of load instruction since the device perform one instruction fetch for a block of threads which is in SIMT manner.

The proposed algorithm is based on the three key concepts that are explained in detail in following subsections.

A. *Tiling*

In CUDA the programmer has to explicitly transfer data from slow low-level GM which is visible by all SMs to a fast high-level shared memory ShM within each SM. Tiling the code is to account for the small ShM capacity. The execution style is based on transferring small amount of data followed by data processing. While transforming the code, it is required to perform proper calculation of effective address of array elements (results) based on the workers identifiers which are the block ID and thread ID. It is required to design an algorithm/mechanism that can be used to apply loop tiling on any CUDA program with proper memory hierarchy optimizations. Tiling is guided by the following steps:

1. Identification of proper tile size to be stored in shared memory based on the limited capacity of ShM per CUDA kernel block based on determining the tile size based on matching overall tile data locality with tShM capacity.
2. Loop transformations and proper identification of range of outer and inner loops.
3. Effective address calculations of the array elements to be accessed within the loop iterations (see coalesced access).
4. Boundary check for avoiding the out of bound array index access.
5. Synchronization among loading of data into ShM, execution of operations, and storing the results back into GM.

B. *Coalesced Global Memory Access*

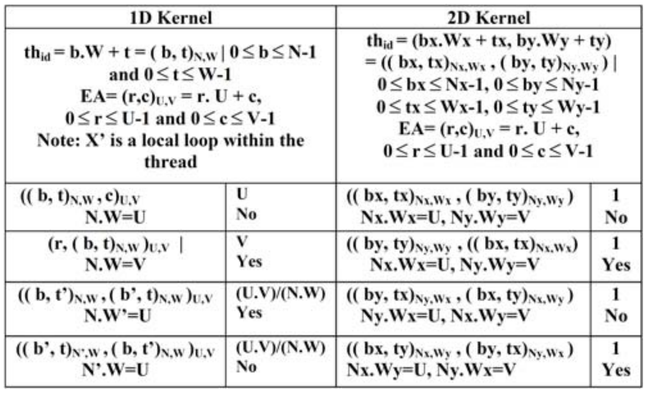
In this section, the objective is to restructure the code so that at execution warps access to GM is done according to a coalesced access pattern to amortize the excessive access cost. Fetching a group of data elements which are stored in distinct memories (coalesced access) is critical to amortize the high cost of accessing GM compared to the speed of the logic. The key idea is to determine all possible mapping

In CUDA a 1-D kernel having NW threads is represented as a set of N blocks each has W elements. To assign some work to each individual thread, each kernel thread is identified by the block b to which it belongs to and some offset t, i.e. $th_{id} = b.W + t$ or as a vector $th_{id} = (b, t)_{N,W}$,where $0 \le b \le N\text{-}1$ and $0 \le t \le W\text{-}1$. Suppose we have a 2-D array of U.V computation results which are stored using row-major scheme as U rows and V columns, the address of the element in row r and column c is $EA= (r,c)_{U,V} = r.\ U + c$, where $0 \le r \le U\text{-}1$ and $0 \le c \le V\text{-}1$. Assigning a thread (worker) to compute a result requires defining a mapping from the thread IDs onto the results so that when the SPMD program is run, each thread uses its own ID in the code to determine the result that it must compute. The mapping of threads IDs onto the result address admits a few possible mapping solutions for $EA = (r,c)_{u,v}$ as computes:

1. $EA = ((b, t)_{N,W}, c)_{U,V} \mid N.W=U$, each thread has one loop to compute V results, no coalesced access,
2. $EA = (r, (b, t)_{N,W})_{U,V} \mid N.W=V$, each thread has one compute U results, coalesced access,
3. $EA = ((b, t')_{N,W}, (b', t)_{N,W})_{U,V} \mid N.W'=U$ and $N'.W=V$, each thread has two loops (denoted by ’) to computes (U.V)/(N.W) results, coalesced access,

4. EA = $((b', t)_{N',W}, (b, t')_{N,W})_{U,V}$ | N'.W=U and N.W'=V, each thread has two loops (denoted by ') to computes (U.V)/(W.N) results, coalesced access.

Note that a coalesced access takes place only when the offset, or second component of EA, is mapped to the thread index, i.e. identified by offset t. The reason is that warps are formed by successive thread IDs for any dimension, i.e. according to row major organization. Table 2 shows the possible mappings of CUDA for 1-D and 2-D kernels (blocks and threads) to a 2-D array of results of size space N.W. Similar approach is used for higher dimension kernels.



1D Kernel		2D Kernel	
$th_{id} = b.W + t = (b, t)_{N,W} \mid 0 \le b \le N-1$ and $0 \le t \le W-1$ EA= $(r,c)_{U,V} = r.U + c$, $0 \le r \le U-1$ and $0 \le c \le V-1$ Note: X' is a local loop within the thread		$th_{id} = (bx.Wx + tx, by.Wy + ty) = ((bx, tx)_{Nx,Wx}, (by, ty)_{Ny,Wy}) \mid$ $0 \le bx \le Nx-1, 0 \le by \le Ny-1$ $0 \le tx \le Wx-1, 0 \le ty \le Wy-1$ EA= $(r,c)_{U,V} = r.U + c$, $0 \le r \le U-1$ and $0 \le c \le V-1$	
$((b, t)_{N,W}, c)_{U,V}$ N.W=U	U No	$((bx, tx)_{Nx,Wx}, (by, ty)_{Ny,Wy})$ Nx.Wx=U, Ny.Wy=V	1 No
$(r, (b, t)_{N,W})_{U,V} \mid$ N.W=V	V Yes	$((by, ty)_{Ny,Wy}, ((bx, tx)_{Nx,Wx})$ Nx.Wx=U, Ny.Wy=V	1 Yes
$((b, t')_{N,W}, (b', t)_{N,W})_{U,V}$ N.W'=U	(U.V)/(N.W) Yes	$((by, tx)_{Ny,Wx}, (bx, ty)_{Nx,Wy})$ Ny.Wx=U, Nx.Wy=V	1 No
$((b', t)_{N',W}, (b, t')_{N,W})_{U,V}$ N'.W=U	(U.V)/(N.W) No	$((bx, ty)_{Nx,Wy}, (by, tx)_{Ny,Wx})$ Nx.Wy=U, Ny.Wx=V	1 Yes

Table 2: Possible 1-D and 2-D Kernel mapping to a 2-D Array of results

C. *Maximizing Utilization*

Within each SM, ShM is partitioned among active blocks which are assigned to SM for simultaneous execution. Therefore the tile sizes must be selected such that the tile data locality that must be loaded into ShM does not constrain the maximum number of active blocks which can be assigned to an SM at a time.

The block size must be chosen greater than or equal to tile size such that each thread in a block loads one element of a tile into ShM. This will reduce instruction fetch and processing overhead of load instruction since the device perform one instruction fetch for a block of threads which is in SIMT manner. On the other hand, too large block sizes must be avoided limiting the number of active blocks per SM. The number of active blocks must be no less than the number of SPs in any given SM to avoid limiting the number of active threads per SM. In the case of such limitation, cores utilization will be reduced because SM maps an active block as an execution unit to each SP. Active Blocks can be calculated using equation (1):

$$Active\,Blocks = \min\left[\begin{matrix} \min\left(\left\lceil \frac{Warp\ per\ SM}{Warp\ per\ Block} \right\rceil, SPs\ per\ SM\right) \\ \min\left(\left\lceil \frac{Shared\ Memory\ per\ SM}{Shared\ Memory\ per\ Block} \right\rceil, SPs\ per\ SM\right) \end{matrix}\right] \rightarrow (1)$$

Here,

$$Warps\ Per\ Block = \frac{Threads\ Per\ Block}{Threads\ Per\ Warp} \rightarrow (1.1)$$

$$Shared\ Memory\ Per\ Block = Tile\ Size \times Data\ Element\ Size$$
$$\times\ Number\ of\ Data\ Elements\ to\ load\ for\ one\ result \rightarrow (1.2)$$

For example, if Threads Per Block = 256, Tile Size = 16 x 16 = 256, Data Element Size = 4 bytes, and Number of Data Elements to load for one result = 2 (as in case of Matrix Multiply) then the Active Blocks = 4. Suppose Warps Per SM = 32, Shared Memory Per SM = 16384, and SPs Per SM = 8. Therefore the number of active blocks that can be handled by an SM at a given time is:

$$Warps\ Per\ Block = \frac{256}{32} = 8$$
$$Shared\ Memory\ Per\ Block = 256 \times 4 \times 2 = 2048$$
$$Active\ Blocks = \min\left[\begin{matrix} \min\left[\left\lceil \frac{32}{8} \right\rceil, 8\right] \\ \min\left[\left\lceil \frac{16384}{2048} \right\rceil, 8\right] \end{matrix}\right] = \min\left[\begin{matrix} \min[4, 8] \\ \min[8. 8] \end{matrix}\right] = \min\left[\begin{matrix} 4 \\ 8 \end{matrix}\right] = 4$$

Since within each SM only one block is assigned to each SP, therefore, the Utilization (U) of SM is determined by the ratio of number of active blocks assigned to an SM over the number of SPs in SM:

$$U = ActiveBlocks\ /\ SPperSM =$$
$$WarpperSM\ /(WarpperBlock * SPperSM)$$

In the previous example the utilization of the GPU will be U=4/8=0.5. In other words, only one SP out of 2 will be active in the computation of the Kernel described in the above example.

In the case the number of warps in a given block exceeds the maximum (k) that can be handled at any given time by the SP, the execution of these warps will be serialized by the SP. Thus each SP will have to run for a number of small cycles (S-Cycles) working in order to complete processing of its assigned block. In each small cycle SP executes k warps while switching from one warp to another whenever there a running warp encounters a high-latency exception such as a load or store dealing with GM. Since the warp executes in SIMD mode, the duration of each small cycle (k warps) depends on the thread granule size and the number of load and store within each thread.

To expose to peak performance, the application threads must be massively and uniformly spread over the SMs so that the only performance saturation comes from mapping the application to the GPU. Furthermore, peak performance will be expected because all the SM and SPs are involved in the execution. To identify the conditions for peak performance, one can analyze the repetition cycles occurs during the kernel execution. Since, there are two levels of kernel block scheduling in the device. The blocks are first scheduled to be executed on each SM and then each SM schedules these blocks to multiple SPs within the SM. We termed the repetitions due to first scheduling as large cycles (L-Cycles) and the repetitions due to second scheduling as small cycles (S-Cycles).

$$L-Cycles = \frac{Kernel\ Blocks\ per\ SM}{SPs\ per\ SM} \rightarrow (2)$$

$$Here,\ Kernel\ Blocks\ per\ SM = Application\ SpaceSize / Tile\ Size$$

$$S-Cycles = \frac{(Warps\ per\ Block \times SPs\ per\ SM)}{Max.Warps\ per\ SM} \rightarrow (3)$$

These repetitions should satisfy the following conditions to achieve peak performance:

1. Divides L-Cycles into integer number of S-Cycles
2. L-Cycles should be greater than or equal to 1
3. S-Cycles should be greater than 1
4. The ratio of L-Cycles to S-Cycles should be minimum

In our experiments for Matrix Multiply, we found the following repetitions (Table 3) at their peak performance. Here, W/B = Warps Per Block, B/SM = Blocks Per SM.

D. Proposed CUDA Restructuring Algorithm

The proposed restructuring algorithm is based on the following steps:

Step 1: *Analyze the granule size in the loop body and the data locality needed and determine thread granule size:*

a. *Thread Granule Size: carry out loop distribution/fusion or statement distribution/fusion to control the thread granule: the number of load/store, number of arithmetic operations, and the needed data locality*
b. *Carry out statement distribution if statement has too many arithmetic operations or requiring too many locality*
c. *Might carry out the opposite of the above steps in the case of too fine granule size of very limited locality*

Step 2: *Tile the resulting loop (or loops) by generating all possible tiled loop arrangements and select one or more tiled arrangements with coalesced memory access.*

Step 3: *Determine the best possible combination of Threads per block (TPB) and the Tile Size(TS) to get the maximum utilization of the machine resources. For thread granule size = 1, TPB = TS. We need to generate all possible TPB and TS, and their respective Warps Per Block (WPB) and Shared Memory Per Block (ShMPB) using the equation (1.1 and 1.2).*

a. *Identify Active Blocks using equation (1) for each of the combination of TPB and TS*
b. *Select the combinations which gives the maximum Active Blocks*

Find out L-Cycles and S-Cycles of the selected combinations. The combination of TPB and TS which satisfies all four conditions of repetitions, as identified in section IV.C, is expected to give the best performance.

IV. Performance Evaluation

A. Non-Coalesced Vs Coalesced Global Memory Access

Figure 2 shows the comparison of application performance in terms of throughput (GFLOPS). There is a significant performance improvement of the application if tile is first loaded into shared memory from global memory with coalesced global memory access and do the computations in shared memory. As in coalesced global memory access, threads in half warp (16 threads) access consecutive memory locations in one cycle so reducing the memory accesses by a factor of (1/16 = 0.0625). In ideal case, the execution time by using coalesced global memory access should reduce to (1-0.0625) * 100 = 93.75%. In our experiments, we have got (1 – 0.11973/0.98727) x 100 = 87.87% reduction in execution time (i.e. improvement in throughput) for N = 262144 which is approximately equal to the ideal with very slight overhead of 5.88%.

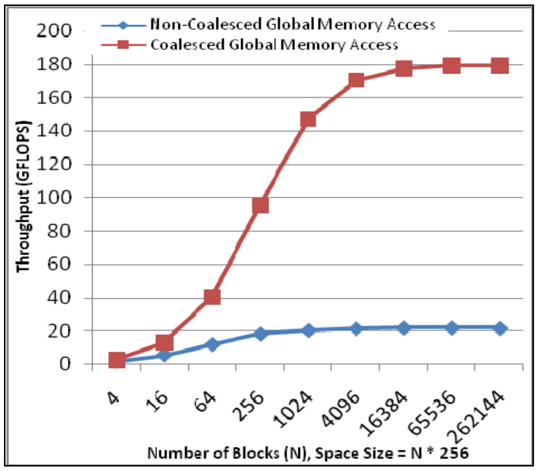


Figure 2: Matrix Multiplication using Shared Memory with (a) Non-Coalesced Global Memory Access and (b) Coalesced Global Memory Access.

B. Computation using Shared Memory Vs Global Memory

Figure 3 shows the performance, in terms of throughput (GFLOPS), comparison of matrix multiplication by computation using global memory and computation using shared memory with non-coalesced load from global memory. Our experimental results as shown in figure 2 and 3 shows that computation using shared memory is faster than the computations using global memory that is loading registers directly from global memory is slower than the loading data from global memory to shared memory and then to registers. It means that the latency of loading data from global memory to registers is much more than the latency from global memory to shared memory plus the latency from shared memory to registers.

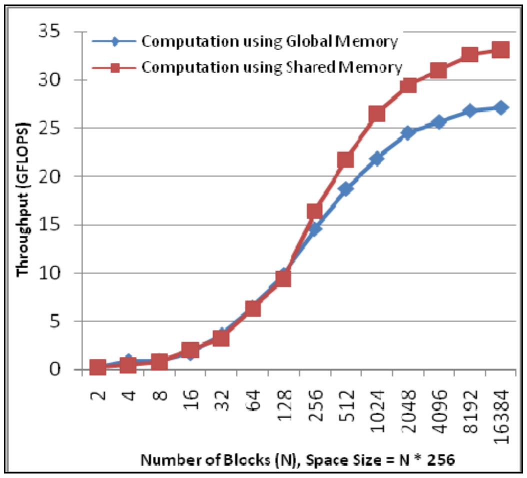


Figure 3: Matrix Multiplication using Computations with (a) Global Memory and (b) Shared Memory

L_{GMR} : *Data Loading Latency from Global Memory to Registers*
L_{GMShM} : *Data Loading Latency from Global Memory to Shared Memory*
L_{ShMR} : *Data Loading Latency from Shared Memory to Registers*

$$L_{GMR} > L_{GMShM} + L_{ShMR}$$

C. Different Thread Granule Size

Figure 4 shows the performance, in terms of throughput (GFLOPS), comparison of matrix scaling application that is multiplying each element of a matrix by an arbitrary value. Since, the ratio of load and compute for this application is 1:1 which means that to do one operation a thread has to load one element. So, increasing thread granule size will not be beneficial. The increase in throughput depends on the application itself, it will not be achieved by increasing the thread granule size. The application which has low ratio of load to computation will give higher throughput than the application which has high value of this ratio.

D. Block Sizes Comparison

Figure 5 shows the application performance comparison in terms of throughput (GFLOPS). The results show that increase in number of threads per block may decrease the performance by restricting the concurrent number of blocks per SM (reducing SM capacity utilization). For block dimension 16 x 16, one block contains 256 threads implies 256/32 = 8 warps/block (Here, 32 is the number of threads per warp) that is one SM can have 32/8 = 4 blocks at a time (Here, 32 is the maximum number of warps per SM). While for block dimension 22 x 22, one block contains 484 threads implies 484/32 = 16 warps/block that is one SM can have 32/16 = 2 blocks at a time. Since, SM maps each block to one SP for execution so in this case the application use only 2 out of 8 SPs per SM that is (2/8)*100 = 25% utilization. Also, in the case, the workload per SP is also

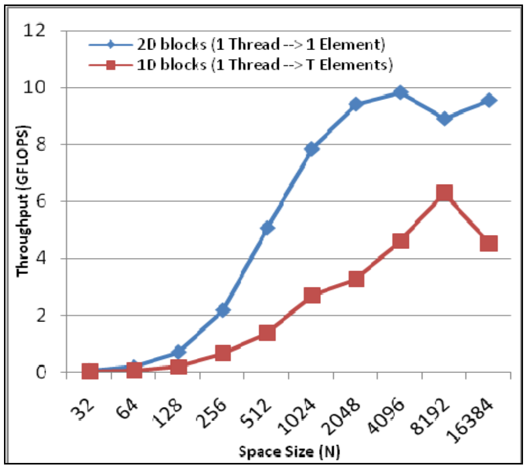


Figure 4: Matrix Scaling with different Thread Granularities

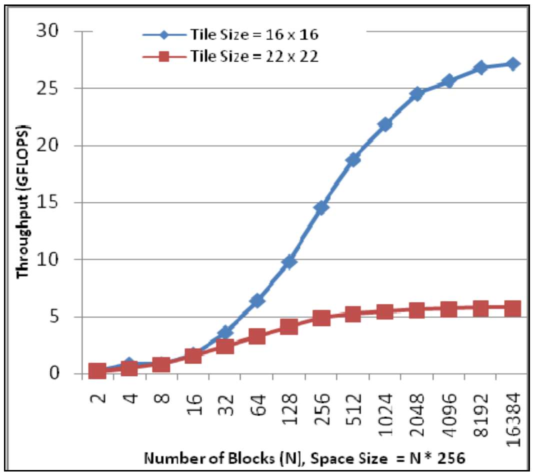


Figure 5: Matrix Multiplication using only global memory with different number of threads per block (a) 16 x 16 = 256 threads/block and (b) 22 x 22 = 484 threads /block

increasing from 8 warps to 16 warps while decreasing the concurrent threads per SM from 1024 to 968. This reduces the overall performance of the application.

E. Memory Usage per Block

Figure 6 shows the performance, in terms of throughput (GFLOPS), comparison of matrix scaling application. In these experiments, we used different tile size to be loaded into shared memory that is different shared memory allocation per block. By using some profiling information provided by the compiler, it seems that there is some compiler overhead (32 bytes) associated in terms of shared memory allocation. Also, the shard memory is allocated to a block in multiples of the shared memory allocation unit that is 512 bytes. For Tile Size = 16 x 16 to load two tiles (one for source and one for target) having each element of 4 bytes, shared memory allocation per block is (16 x 16 x 2 x 4) + 32 = 2080 bytes implies actual shared memory allocation = 2560 bytes that is 512 x 5. Since, the shared memory is partitioned among the blocks per SM so in the case the concurrent blocks per SM is 16384/2560 = 6.4 implies 6 blocks/SM but due to maximum warps limit (32 warps/SM) only 4 blocks (1 block = 8 warps) will be scheduled to one SM at a time. For Tile Size = 32 x 32, shared memory allocation per block is (32 x 32 x 2 x 4) + 32 = 8224 bytes implies actual shared memory allocation = 8704 bytes that is 512 x 17. So in this case the concurrent blocks per SM is 16384/8704 = 1.88 implies only 1 block/SM will be scheduled to one SM at a time. This reduces the SM capacity utilization which in response reduces the overall performance of the application. The results also show that the computation using shared memory saturates the device at number of blocks = 512 in the application that is throughput decreases after this threshold. Here, the peak throughput achieved at N = 8192 and TPB =16 that is 8192/16 = 512 blocks. We may have the same pattern for TPB = 32 if use larger space size (N) but we could not run our experiments for N > 16384 due to

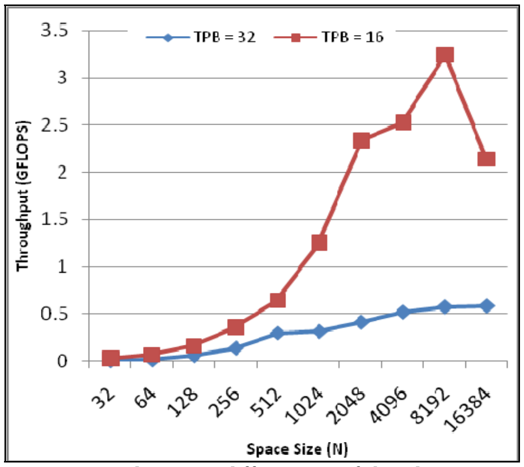


Figure 6: Matrix Scaling using different size of shared memory per block (a) TPB = 32, 32 x 32 x 2 x 4 = 8KB and (b) TPB = 16, 16 x 16 x 2 x 4 = 2 KB

limited global memory of 4 GB. For N = 16384 (2^{14}) to load two matrices having each element of 4 bytes, $2^{14 \times 2} \times 2 \times 2^2 = 2 \times 2^{30}$ = 2 GB < 4 GB while for N = 32768 (2^{15}), $2^{15 \times 2} \times 2 \times 2^2 = 8 \times 2^{30}$ = 8 GB > 4 GB.

V. CONCULSION

We presented a restructuring algorithm to optimize a CUDA program based on three key concepts: Tiling, Coalesced Global Memory Access and Maximizing Utilization. The execution model of synthesized kernel consists of uniformly distributing the kernel threads to keep all cores busy while transferring a tailored data locality which is accessed using coalesced pattern to amortize the long latency of the secondary memory. In the evaluation, we implement some simple applications using the proposed restructuring strategy and evaluated the performance in terms of execution time and GPU throughput. Obtained results were analyzed in view of proposed optimization parameters which reinforces the proposed restructuring.

ACKNOWLEDGMENT

We thank King Fahd University of Petroleum and Minerals and the Department of Information and Computer Science for access to its GPU computers.

REFERENCES

[1] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-lite: Reducing GPU programming complexity. International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2008.

[2] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann, OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization, Proc. 14th ACM SIGPLAN Symp. on Prin. and Prac. of Parallel Programming, 2009.

[3] Tianyi David Han and Tarek S. Abdelrahman, “hiCuda: A high-level Directive-based Language for GPU Programming”, GPGPU’09, March 8, 2009.

[4] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kr uger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 26(1):80-113, March 2007.

[5] K. Mueller, F. Xu, and N. Neophytou. Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography? SPIE Electronic Imaging 2007, Computational Imaging , Keynote, 2007.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Kevin Skadron, “A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA”, in The First Workshop on General Purpose Processing on Graphics Processing Units, 2007.

[7] R. Belleman, J. Bedorf, S.P. Zwart, High performance direct gravitational N-body simulations on graphics processing units – II: an implementation in CUDA, New Astronomy 13 (2) (2008) 103–112.

[8] Gabe Rudy, “CUDA-CHiLL: A Programming Language Interface for GPGPU Optimizations And Code Generation”, MS Thesis, School of Computing, University of Utah, USA, August 2010.

[9] Asanovic K., Bodik R., Demmel J., Keaveny T., Keutzer K., Kubiatowicz J., Morgan N., Patterson D., Sen K., Wawrzynek J., Wessel D., Yelick K.: “A View of Parallel Computing Landscape”, Communications of ACM 52(10) (2009) 56-67.

[10] David B. Kirk and Wen-mei W. Hwu, “Programming Massively Parallel Processors: A Hands-on Approach”, Published by Elsevier Inc. ISBN: 978-0-12-381472-2, 2011.