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 resource optimization. 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 pair-wise 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 host 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 speedups 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 OpenMP constructs into kernel with default 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 resource optimization. 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. 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 II presents some analysis of GPU that is critical for performance tuning. Section III presents a proposed approach for restructuring algorithm for CUDA. Section IV presents an example of applying the proposed strategy to develop and optimized kernel for matrix multiplication. Section V presents the evaluation of applications and comments on execution times and throughput. Section VI presents the comparison of proposed strategy with other approaches. Finally, Section VII 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. Each GPU currently comes with up to 4 GB of graphics double data rate (GDDR) DRAM referred to as global memory (GM) that is visible to all threads in all blocks. Each SM 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 7000	Tesla C2070	Tesla C2075
GM	4 GB	6 GB	1.5 GB	6 GB
Total SM	30	16	14	14
SP per SM	8	32	32	32
Total Cores	30 * 8 = 240	16 * 32 = 512	14 * 32 = 448	14 * 32 = 448
ShM/B	16 KB	48 KB	48 KB	48 KB
Reg/SM	2^{14}	2^{15}	2^{15}	2^{15}
Warp Size	32	32	32	32
Max th/B	2^{9}	2^{10}	2^{10}	2^{10}
Max B dim	2^{9}x 2^{9}x2^{6}	2^{10}x 2^{10}x2^{6}	2^{10}x 2^{10}x2^{6}	2^{10}x 2^{10}x2^{6}
Max grid dim	2^{16}x 2^{16}x1	2^{16}x 2^{16}x2^{16}	2^{16}x 2^{16}x2^{16}	2^{16}x 2^{16}x2^{16}
Clock Rate	1.3 GHz	1.3 GHz	1.15 GHz	1.15 GHz
Warps/SM	32	48	48	48
Max. Th/SM	1024	1536	1536	1536
B/SM	8	8	8	8
L2 Cache	No	Yes	No	Yes

Table 1: Some features for some NVIDIA 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 or more 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-threads instruction called warps, where each warp is being executed in the single-instruction multiple-data (SIMD) manner. A warp takes multiple cycles for computation instructions due to the limited number of functional units (SPs) within SM.

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].

Fig. 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] while individual threads will be distributed among multiple SPs within the SM. An SM can handle at most 8 blocks at a time. 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 III. For many GPUs (Table 1), each SM can handle 32 warps at a time. In Tesla C2070, each SM has 32K registers and 48KB of shared memory.

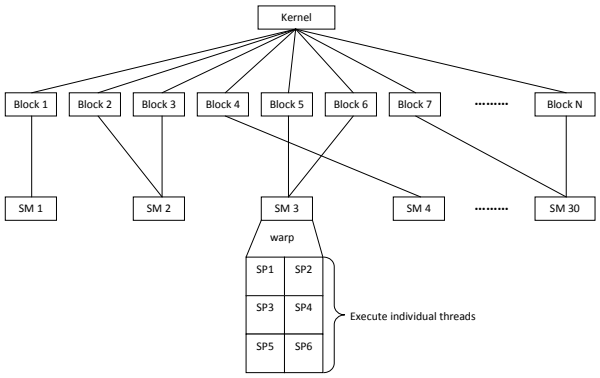

Figure 1: CUDA Kernel Execution Hierarchy

SM manages thread ids and threads execution. Threads within a block cooperate 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 III.

Each SM schedules one warp at a time with zero overhead warp 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. 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 SPs [10].

GM is partitioned into segments of size equal to 32, 64 or 128 bytes 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 active warp per SM.

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 less than or equal to tile size such that each thread in a block loads one or more elements 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 and matching overall tile data locality with ShM 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 $0 \le b \le N-1$ and $0 \le t \le W-1$. 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 $0 \le r \le U-1$ and $0 \le c \le V-1$. 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. $EA = ((b', t)_{N',W}, (b, t')_{N,W})_{U,V} \mid N'.W=U$ 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 with corresponding tile size (upper parameter) and coalesced (Yes) or non-coalesced (No) accesses. 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	$((by, ty)_{Ny,Wx}, (bx, ty)_{Nx,Wy})$ $Ny.Wx=U, Nx.Wy=V$	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

For example, assume a 2-D(U,U) array res() of results, and TxT as being the tile size. Let's use a 1D kernel defined by $th_{id} = (b, t)_{N,W}$. For 1-D kernel, we may use the solution shown in the third row of Table 2. The corresponding constraints leads to N=U/T blocks and each block has each W=T threads. The effective address of a result res() is EA = (b*T+t')*U + b'*T+t. Each kernel thread consists of a double nested loop, where the outer loop (t': U/T iterations) and inner loop (b': T iterations). It is clear that access is coalesced because t is in the least significant position. This solution is also implemented and evaluated in the performance evaluation (Section IV, Fig. 5).

C. Resource Optimization

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 less than or equal to tile size such that each thread in a block loads one or more elements 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 due to large number of warps per block. The number of active warps must be no less than the maximum warps per SM (for full occupancy) in any given SM to avoid limiting the number of active threads per SM. Active Blocks can be calculated using equation (1).

$$Active\,Blocks = \min\left[\begin{array}{c} \min\left(\left\lfloor \frac{Warp\ per\ SM}{Warp\ per\ Block} \right\rfloor, Max.\,Blocks\ per\ SM\right) \\ \min\left(\left\lfloor \frac{Shared\ Memory\ per\ SM}{Shared\ Memory\ per\ Block} \right\rfloor, Max.\,Blocks\ per\ SM\right) \end{array}\right] \rightarrow (1)$$

Here,

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

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

$$Warps\ Per\ Block = \frac{256}{32} = 8$$

$$Shared\ Memory\ Per\ Block = 256 \times 4 \times 2 = 2048$$

$$Active\ Blocks = \min\left[\begin{array}{c} \min\left[\left\lfloor \frac{32}{8} \right\rfloor, 8\right] \\ \min\left[\left\lfloor \frac{16384}{2048} \right\rfloor, 8\right] \end{array}\right] = \min\left[\begin{array}{c} \min[4, 8] \\ \min[8. 8] \end{array}\right] = \min\left[\begin{array}{c} 4 \\ 8 \end{array}\right] = 4$$

For example, if Threads per Block is 256, Tile Size is 256, Data Element Size is 4 bytes, and Number of Data Elements to load for one result is 2, then the Active Blocks is 4. Suppose Warps Per SM is 32, Shared Memory Per SM is 16384, and Max. Blocks Per SM is 8. Therefore the number of active blocks that can be handled by an SM at a given time can be calculated using eq. (1).

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 and threads scheduling in the device. The blocks are first scheduled to be executed on each SM and then each SM schedules the individual threads within a block to multiple SPs within the SM based on selecting one warp at a time. The repetitions (or serialization affect) due to first scheduling can be analyzed as average kernel blocks per SM and the repetitions due to second scheduling as small cycles (S-Cycles) which occurs due to limited number of SPs (Thread Processors) that can execute one thread at a time.

$$Average\ Kernel\ Blocks\ per\ SM\ (AKBPSM) = \frac{Total\ Kernel\ Blocks}{Total\ SMs} \rightarrow (2)$$

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

$$S - Cycles = \frac{\left(Active\ Blocks \times Threads\ Per\ Block\right)}{SPs\ per\ SM} \rightarrow (3)$$

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

1. Both AKBPSM and S-Cycles should be greater than or equal to 1.
2. S-Cycles should be an integer value to balance the threads among multiple SPs.
3. S-Cycles should be as large as possible.
4. AKBPSM should be the least possible to minimize serialization.

In our experiments for Matrix Multiply, we found the following repetitions (Table 3) at their peak performance. Here, TPB = Threads Per Block, TS = Tile Size, AB = Active Blocks, TKB = Total Kernel Blocks, Exec. Time = Execution Time in seconds.

Tesla C2070 Machine (N = 2048 x 2048)						
TPB	TS	AB	TKB	S-Cycles	AKBPSM	Exec. Time
512	2048	3	2048	48	146.28	2.45
512	1024	3	4096	48	292.57	2.47
256	1024	6	4096	48	292.57	2.51
512	512	3	8192	48	585.14	2.53
256	512	6	8192	48	585.14	2.55
256	256	6	16384	48	1170.28	2.62

Table 3: Repetitions Analysis of Matrix Multiplication for Resource Optimization

We performed similar analysis on other applications including different implementations of Matrix Transpose available with CUDA SDK, Matrix Scaling and found similar trend of execution time.

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 optimal distribution of blocks and threads among SMs and SPs respectively. 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. *Calculate S-Cycles for each of the combinations using equation (3) and select the combinations that have the maximum value.*
c. *Calculate AKBPSM for the selected combinations and the one that has the minimum value of AKBPSM will give the best performance.*

IV. EXAMPLE

In this section, we will show the working steps of writing a matrix multiplication application from the sequential code (Code Listing 1, for N x N matrices) to optimized CUDA kernel.

```
void matrix_multiply(float **C, float **B, float **A, int N)
{
   for(int ty=0; ty < N; ty++)
      for(int tx=0; tx < N; tx++){
         C[i][j] = 0;
            for(int k=0; k < N; k++)
              C[i][j] += A[i][k] * B[k][j];
      }
}
```

Code Listing 1: Matrix Multiplication Sequential Code

Step 1: due to the limited data locality and few arithmetic operations in the statement, each thread can simply focus on calculating one resultant element that is thread granule size = 1.

```
void tiled_matrix_multiply(float **C, float **B, float **A, int N)
{
   for(int by=0; by < N; by+=TILE_Y)
      for(int bx=0; bx < N; bx+=TILE_X)
         for(int ty=0; ty < TILE_Y; ty++)
            for(int tx=0; tx < TILE_X; tx++)
              for(int bk=0; bk < N; bk+=TILE_X)
                 for(int k=0; k < TILE_X; k++)
                  C[by+ty][bx+tx] = A[by+ty][bk+k] * B[bk+k][bx+tx];
}
```

Code Listing 2(a): Matrix Multiplication Tiled Version

```
__global__ void
tiled_matrix_multiply(float *C, float *B, float *A, int N)
{
   int by = blockIdx.y * TILE_Y;
   int bx = blockIdx.x * TILE_X;
   int ty = threadIdx.y;
   int tx = threadIdx.x;

   for(int bk=0; bk < N; bk+=TILE_X)
      for(int k=0; k < TILE_X; k++)
         C[(by + ty) * N + bx + tx] = A[(by + ty) * N + bk + k]
                                      * B[(bk + k) * N + bx + tx];
}
```

Code Listing 2(b): Matrix Multiplication CUDA kernel

```
__global__ void
coalesced_matrix_multiply(float *C, float *B, float *A, int N)
{
   int by = blockIdx.y * TILE_Y;
   int bx = blockIdx.x * TILE_X;
   int ty = threadIdx.y;
   int tx = threadIdx.x;

   float Csub=0;
   __shared__ float As[TILE_Y][TILE_X];
   __shared__ float Bs[TILE_X][TILE_X];

   for(int bk=0; bk < N; bk+=TILE_X){
      As[ty][tx] = A[(by + ty) * N + bk + tx];
      Bs[ty][tx] = B[(bk + ty) * N + bx + tx];

      __syncthreads();

      for(int k=0; k < TILE_X; k++)
         Csub += As[ty][k] * Bs[k][tx];
   }

   __syncthreads();

   C[(by + ty) * N + bx + tx] = Csub;
}
```

Code Listing 3: CUDA kernel with coalesced memory accesses

Step 2: Code Listing 2(a) shows the tiled version of Code Listing 1 by using general strategy of loop tiling for uniprocessors that is split each loop of a nested loop-set into a pair of adjacent loops in the loop nest, with the outer loop (tiling loop) traversing tiles (blocks), and the inner loop (intra-tile loop) covering the iteration points within the

tile. Code Listing 2(b) shows the corresponding CUDA kernel implementation using 2D blocks and threads that maps the outer four loops of Code Listing 2(a) to the blocks and threads dimensions in Code Listing 2(b). At this stage, accessing to matrix C and B are satisfying the mappings of coalesced memory access as shows in second row of 2D kernel mappings in Table 2 while access to matrix A is not coalesced.

Code Listing 3 shows the modified kernel to perform coalesced loads of matrix A and B using shared memory and coalesced stores to the resultant matrix C. Here, we are assuming the same dimensions for thread blocks and matrix tiles. We also need to add barrier synchronization among threads of the same block using __syncthreads() between tiles load and compute statement within the traversal of all tiles of matrices A and B. Also a barrier is required before storing the resultant tile of matrix C due to difference in the traversal order of load/store and computation statements.

Step 3: For Tesla C2070 using the resource optimization strategy as explained in section III.C, we found optimal values for threads per block and tile sizes as TPB = 32 * 16 512 and TS = 32 * 64 = 2048. Code Listing 4 shows the modified kernel of Code Listing 3 to handle the case of TPB < TS, for this we need to add loop for each load, compute and store statement to correctly load the whole tile, compute the results, and store the whole resultant tile to the destination.

```
__global__ void
gen_coalesced_matrix_multiply(float *C, float *B, float *A, int N)
{
   int by = blockIdx.y * TILE_Y;
   int bx = blockIdx.x * TILE_X;
   int ty = threadIdx.y;
   int tx = threadIdx.x;

   float Csub[TILE_Y/BLOCK_Y];
   __shared__ float As[TILE_Y][TILE_X];
   __shared__ float Bs[TILE_X][TILE_X];

   for(int bk=0; bk < N; bk+=TILE_X){
      for(int i=0; i < TILE_Y/BLOCK_Y; i++){
        As[ty + i * BLOCK_Y][tx] = A[(by + ty + i * BLOCK_Y)
                                          * N + bk + tx];
      }
      for(int i=0; i < TILE_X/BLOCK_Y; i++){
       Bs[ty + i * BLOCK_Y][tx] = B[(bk + ty + i * BLOCK_Y)
                                          * N + bx + tx];
      }

      __syncthreads();

      for(int i=0; i < TILE_Y/BLOCK_Y; i++)
        for(int k=0; k < TILE_X; k++)
            Csub[i] += As[ty + i * BLOCK_Y][k] * Bs[k][tx];
   }

   __syncthreads();

  for(int i=0; i < TILE_Y/BLOCK_Y; i++)
     C[(by + ty + i * BLOCK_Y) * N + bx + tx] = Csub[i];
}
```

Code Listing 4: Optimized CUDA Kernel

V. Performance Evaluation

A. *Non-Coalesced Vs Coalesced Global Memory Access*

Fig. 2 shows the GPU throughput (GFLOPS) of two 2D kernel mapping solutions for the matrix multiply. These solutions correspond to a tiled loop with and without coalesced GM access which are illustrated in the 2nd columns of Table 2 at the 2nd (No) and 3rd (Yes) rows, respectively. According to the solution (3rd row), a tile is first loaded into ShM from GM using a coalesced access and do the computations while data is in ShM. 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 an ideal factor of 94%. The above solution allows reducing the program execution time by 87.87%.

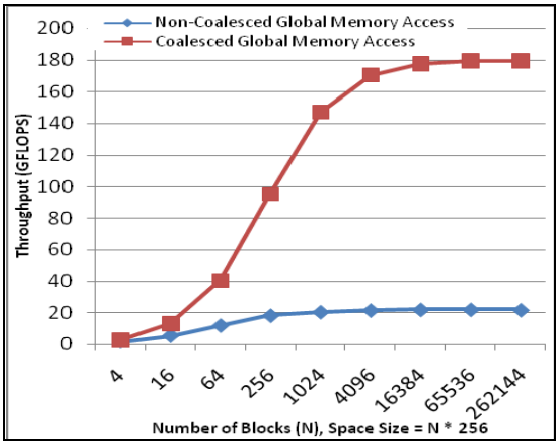

Figure 2: Matrix Multiplication using Shared Memory with (a) Non-Coalesced Global Memory Access and (b) Coalesced Global Memory Access.

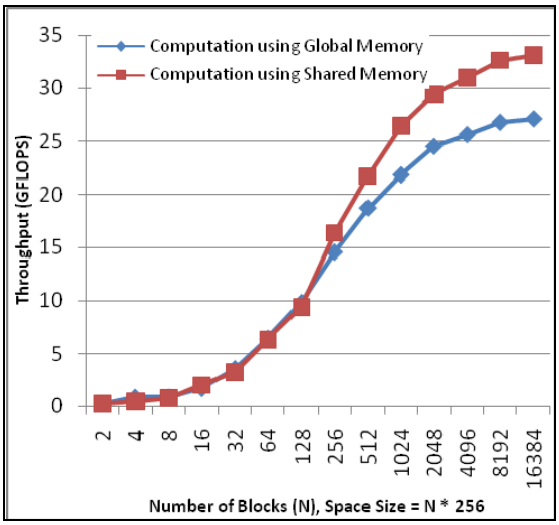

Figure 3: Matrix Multiplication using Computations with (a) Global Memory and (b) Shared Memory

Even with no coalesced GM memory access, copying a tile from GM onto ShM before execution is faster (about 22%) than loading the SM registers directly from GM. Fig. 3 shows the corresponding throughput (GFLOPS):

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

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

B. *Block Sizes Comparison*

We refer to Resource Optimization described in Section III. Increasing the number of threads per block may decrease the performance due to restriction in the concurrent number of blocks per SM which reduces SM capacity utilization.

A 256-thread block (option 1) has 8 warps each has 32 threads. Thus each SM will be assigned 4 blocks at a time. While a 484-thread block (option 2) has 16 warps leading each SM to be assigned 2 blocks at a time. Comparing the above two options, it is clear that option 1 provides larger S-cycles and smaller AKBPSM. Here, option 1 represents a case for the best possible resource utilization. Fig. 4 shows the GPU throughput (GFLOPS) for both options in which the solution corresponding to option 1 is more than 5 times faster than that corresponding to option 2.

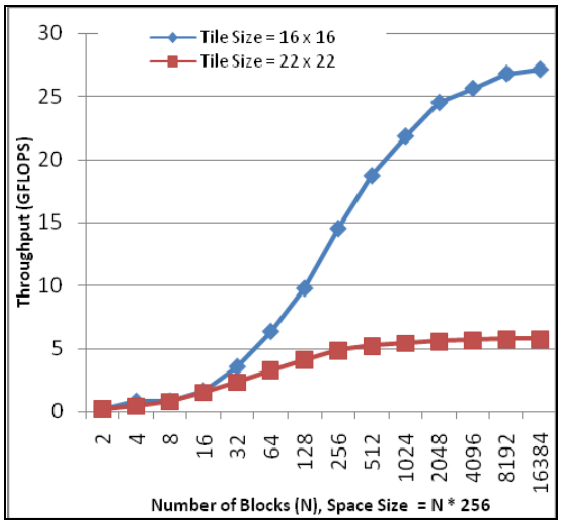

Figure 4: Matrix Multiplication using only global memory with different number of threads per block (a) 16 x 16 = 256 threads/block and (b) 22 x 22 = 484 threads /block

C. *Memory Usage per Block*

We refer to ShM allocation that was described in Resource Utilization of Section II. Here we use different tile size to be loaded into shared memory and use different ShM allocation per block. Run-time profiling indicates the existence of some compiler overhead associated to each ShM allocation. ShM is allocated in multiples of basic 512 bytes.

A 16x16 tile leads to load a source tile and a result tile requires the allocation of 2080 bytes into ShM including the overhead. The actual ShM allocation is 2560.

Since the shared memory is partitioned among the blocks per SM so in the case of Quadro FX 5800, the concurrent blocks per SM is 16384/2560 = 6.4 implies 6 blocks per SM. As only 32 warps are assigned to SM, only 4 blocks will be scheduled to one SM at a time.

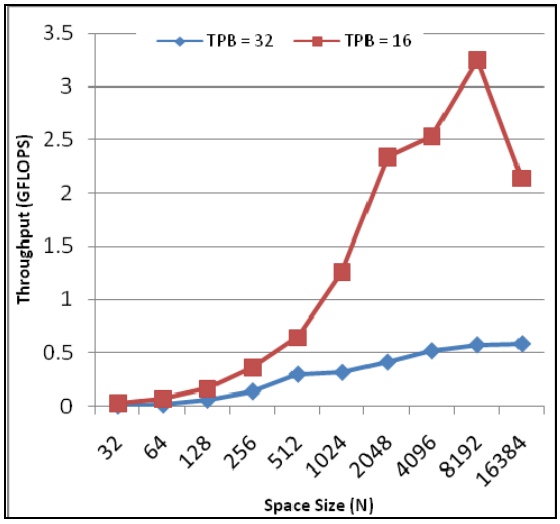

Figure 5: Matrix Scaling using different size of shared memory per block (a) TPB = 32, 32 x 32 x 2 x 4 = 8KB and (b) TPB = 16, 16 x 16 x 2 x 4 = 2 KB

A 32x32 tile requires ShM allocation per block that is 8224 bytes which implies that the actual ShM allocation is 8704 bytes. In this case the concurrent blocks per SM are 1.88. Thus only 1 block per SM will be scheduled at a time. This reduces the SM capacity utilization which in response reduces the overall performance of the application. Fig. 5 shows the throughput of matrix scaling using different size of shared memory per block. 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 512 blocks. We may have the same pattern for TPB=32 if we use larger space size (N) but we could not run our experiments for N > 16384 due to GM limitation to 4 GB. For N = 16384, the load of two matrices requires 2 GB while 8 GB for N = 32768.

VI. Application Results Comparison

A. *Matrix Multiplication*

Tesla C2070 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	512	2048	3	2048	48	146.2857143	2.4486
NVIDIA SDK	256	256	6	16384	48	1170.285714	2.6268
CUDALite	32	1024	1	4096	1	292.5714286	21.2396

Table 4: Parameters comparison of different implementations of Matrix Multiplication

We have analyzed the structure of matrix multiplication kernels using CUDALite [1] approach and NVIDIA SDK approach [10]. Both of these implementations used arbitrary values for defining threads per block (TPB) and tile size (TS) which are not optimal values in terms of

resource utilization as we have explained in section III.C. In CUDALite, each thread work on the entire row of the tile resulting in very few threads per block (TPB = 32 as shown in Table 4 that only 1 warp per block) which is not sufficient to hide latency of the global memory transfers. Also, in CUDALite, a tile allocation is also done for results which causes large shared memory usage per thread block that restricts the number of Active Blocks (AB = 1, see Table 4, can be calculated using eq. (1)) that highly reduces the S-Cycles to 1. In NVIDIA SDK approach, 2D thread blocks of 16 x 16 dimensions is defined with same tile sizes so each thread work on one element of each tile but these values produces large number of average kernel blocks per SM which causes increased overhead of blocks allocation and thus limited performance. The optimal value of TPB and TS for Tesla C2070 GPU are 512 and 2048 respectively as proposed by our restructuring algorithm (see Table 4) and gives the minimum execution time in comparison of the other approaches.

B. Matrix Scaling

Tesla C2070 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	512	4096	3	1024	48	73.14285714	0.0014
CUDALite	32	1024	1	4096	1	292.5714286	0.0096

Table 5: Parameters comparison of different implementations of Matrix Scaling

We have also analyzed the matrix scaling kernel shown as an example in CUDALite [1] paper. We have found similar problems of limited number of active blocks due to large shared memory usage and also large number of average kernel blocks per SM due to small number of threads per blocks as explained in the previous section V.A in the case of matrix multiplication. The optimal value of TPB and TS for Tesla C2070 GPU are 512 and 4096 respectively as proposed by our restructuring algorithm (see Table 5) and gives the minimum execution time in comparison of the CUDALite approach.

C. Matrix Transpose

NVIDIA provides optimized kernels of matrix transpose by analyzing the architectures of shared memory and global memory. In these optimizations, tiles are allocated in shared memory in such a way that the access to the shared memory by different threads at the same time should be free from shared memory bank conflicts. Furthermore, access to global memory by concurrent thread blocks will be done in different partitions of global memory to load the tile from the source matrix and store the tile into transposed matrix. We have applied our resource optimization strategy to two different matrix transpose kernels as provided in NVIDIA SDK. TPB = 512 is obtained as an optimal value for threads per block that maximize S-Cycles (see Table 6 and 7) and hence minimize the execution time in comparison of the defined parameters in NVIDIA documentation.

Quadro FX 7000 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	512	1024	3	4096	48	256	0.0776
NVIDIA SDK	256	1024	5	4096	40	256	0.1084

Table 6: Parameters comparison of Matrix Transpose kernels with no shared memory bank conflicts

Quadro FX 7000 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	512	1024	3	4096	48	256	0.0800
NVIDIA SDK	256	1024	5	4096	40	256	0.1234

Table 7: Parameters comparison of Matrix Transpose kernels with diagonal tiles mapping to blocks to avoid partition camping

VII. CONCULSION

We presented a restructuring algorithm to optimize a CUDA program based on three key concepts: (1) tiling, (2) coalesced global memory access, and (3) resource optimization. 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 to outline some features of 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 and alleviate the tedious task of finding an optimized solution based manually optimizing many parameters. We have also compared our strategy with other implemented approaches of matrix multiplication, matrix scaling, and matrix transpose kernels mentioned in CUDALite and NVIDIA SDK. We found that none of the approach defines a strategy for defining optimal number of threads per block and tile size while our resource optimization strategy helps to determine the optimal values of these parameters that maximize the performance in comparison of the other approaches.

ACKNOWLEDGMENT

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

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.