Exploration of Automatic Optimization for CUDA Programming

Mayez Al-Mouhamed and Ayaz ul Hassan 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. 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 that run multiple threads in parallel and provide memory latency hiding by overlapping long-latency loads in stalled threads with useful computation in other threads [5]. The Compute Unified Device Architecture (CUDA) is an extension to C programming for NVIDIA GPUs. However, porting applications to CUDA remains a non-trivial task even for expert programmers. CUDA programmers have to write GPU code in separate functions with explicit management of data transfer between the host and GPU memories and manual optimization of the GPU memory hierarchy [3].

Performance study of general-purpose GPU programming have been reported [11] for applications such as SRAD structured grid, back-propagation unstructured grid, data encryptions standard, Needleman – Wunsch dynamic programming, and k-means data mining. A CUDA implementation for the gravitational N-body simulations using GPU is reported [12]. 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.

Software tuning of high-performance kernels [6] for GPUs is critical for efficiently running linear solver algorithms such as the Basic Linear Algebra Subprograms (BLAS) kernels. Optimizing programs using the Vector blocking techniques [7] over hybrid architectures (multicore and GPU) proved to be useful for improving performance of the matrix multiply routine (GEMM). Orders of magnitude acceleration is reported compared to multicore without GPU accelerators when architecture and algorithm-specific optimizations are used for implementing dense linear algebra solvers such as the MAGMA library [8]. A three-step optimization is proposed for the QR factorization [9]. QR is factorized as a sequence of tasks with chosen granularity. The kernel for each task is designed. Finally, static scheduling is used when a priori knowledge is available. Otherwise, dynamic scheduling is used by managing data availability and coherency. The reported performance is very close to that obtained using Linear Programming with some limited portability. The implementation complements kernels already available in the MAGMA library.

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], an experimental enhancement to CUDA, allows programmers to write CUDA kernels by using only global memory and automatically transform it in an optimize CUDA kernel to leverage the complex memory hierarchy. Authors claim the tool produces code with performance comparable to hand-coded versions.

Lee et. al have proposed a framework for source-to-source translation of standard OpenMP applications into

CUDA-based code [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. 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 (CUDACHiLL) [13] 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.

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 a proposed approach for restructuring algorithm for CUDA. Section III presents an example of applying the proposed strategy to develop and optimized kernel for matrix multiplication. Section IV presents the comparison of proposed strategy with other approaches. Finally, Section V concludes about this work.

II. 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). The proposed restructuring algorithm aimed at generating efficient CUDA kernels. It 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 $0 \leq b \leq N\text{-}1$ and $0 \leq t \leq W\text{-}1$. 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 $0 \leq r \leq U\text{-}1$ and $0 \leq c \leq V\text{-}1$. 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 = $((b, t)_{N,W}, c)_{U,V}$ | N.W=U, each thread has one loop to compute V results, no coalesced access,
2. EA = $(r, (b, t)_{N,W})_{U,V}$ | N.W=V, each thread has one compute U results, coalesced access,
3. 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)/(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 1 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 = $(b, t)_{N,W}$ | $0 \le b \le$ 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) = $((bx, tx)_{Nx,Wx}, (by, ty)_{Ny,Wy})$ | $0 \le bx \le$ 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**
$(r, (b, t)_{N,W})_{U,V}$ | 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 1: 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 1. 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.

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.

$$Active\ Blocks = \min\begin{bmatrix} \min\left(\left\lceil \dfrac{Warp\ per\ SM}{Warp\ per\ Block}\right\rceil, Max.\ Blocks\ per\ SM\right) \\ \min\left(\left\lceil \dfrac{Shared\ Memory\ per\ SM}{Shared\ Memory\ per\ Block}\right\rceil, Max.\ Blocks\ per\ SM\right)\end{bmatrix} \rightarrow (1)$$

Here,

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

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

$$Warps\ Per\ Block = \frac{256}{32} = 8$$
$$Shared\ Memory\ Per\ Block = 256 \times 4 \times 2 = 2048$$

$$Active\ Blocks = \min\begin{bmatrix} \min\left[\left\lceil \dfrac{32}{8}\right\rceil, 8\right] \\ \min\left[\left\lceil \dfrac{16384}{2048}\right\rceil, 8\right]\end{bmatrix}$$
$$= \min\begin{bmatrix} \min[4, 8] \\ \min[8. 8]\end{bmatrix}$$
$$= \min\begin{bmatrix} 4 \\ 8\end{bmatrix} = 4$$

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

Here, Total Kernel Blocks = Application SpaceSize / Tile Size

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

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).

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. 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 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 occur due to limited number of SPs (Thread Processors) that can execute one thread at a time.

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.

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.*

III. 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

```
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

```
__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

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.

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 1 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 II.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.

IV. Application Results Comparison

A. *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 II.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 2 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 2, 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 and gives the minimum execution time in comparison of the other approaches.

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 II.C 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 3) and gives the minimum execution time in comparison of the CUDALite approach.

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 2: Parameters comparison of different implementations of Matrix Multiplication

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 3: Parameters comparison of different implementations of Matrix Scaling

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 4 and 5) 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 4: 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 5: Parameters comparison of Matrix Transpose kernels with diagonal tiles mapping to blocks to avoid partition camping

V. CONCLUSION

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. 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 using brute-force approach. 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. Currently some frameworks provide auto-tuning of kernel parameters using brute-force approach but 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

Thanks to the ICS-KFUPM and KAUST for giving access to their GPU computers and workstations.

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] Demmel, J. et. al., Self-Adapting Linear Algebra Algorithms and Software, Proc. of the IEEE, Vol. 93, No 2, pp.293-312, 2005.

[7] Volkov, V.; Demmel, J.W., Benchmarking GPUs to tune dense linear algebra, Inter. Conf. on High Performance Computing, Networking, Storage ans Analysis (SC 2008), pp. 1-11, 2008.

[8] Tomov, S.; Nath, R.; Ltaief, H.; Dongarra, J., Dense linear algebra solvers for multicore with GPU accelerators, IEEE Inter. Symp. On Parallel and Distrib. Processing, pp. 1-8, 2010.

[9] Agullo, E. et. Al., QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators, IEEE Inter. Parallel & Distributed Processing Symposium (IPDPS), pp. 932-943, 2011.

[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.

[11] 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.

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

[13] Gabe Rudy, "CUDA-CHiLL: A Programming Language Interface for GPGPU Optimizations And Code Generation", 2010, http://books.google.com.sa/books?id=vg66ZwEACAAJ