CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3591

PAPER

Demystifying the 16 × 16 thread-block for stencils on the GPU

Siham Tabik[1,*,†], Maurice Peemen[2], Nicolas Guil[1] and Henk Corporaal[2]

[1]*Department of Computer Architecture, University of Málaga, 29071, Málaga, Spain*
[2]*Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands*

SUMMARY

Stencil computation is of paramount importance in many fields, in image processing, structural biology and biomedicine, among others. There exists a permanent demand of maximizing the performance of stencils on state-of-the-art architectures, such graphics processing units (GPUs). One of the important issues when optimizing these kernels for the GPU is the selection of the best thread-block that maximizes the overall performance. Usually, programmers look for the optimal thread-block configuration in a reduced space of square thread-block configurations or simply use the best configurations reported in previous works, which is usually 16 × 16. This paper provides a better understanding of the impact of thread-block configurations on the performance of stencils on the GPU. In particular, we model locality and parallelism and consider that the optimal configurations are within the space that provides: (1) a small number of global memory communications; (2) a good shared memory utilization with small numbers of conflicts; (3) a good streaming multi-processors utilization; and (4) a high efficiency of the threads within a thread-block. The model determines the set of optimal thread-block configurations without the need of executing the code. We validate the proposed model using six stencils with different halo widths and show that it reduces the optimization space to around 25% of the total valid space. The configurations in this space achieve at least a throughput of 75% of the best configuration and guarantee the inclusion of the best configurations. Copyright © 2015 John Wiley & Sons, Ltd.

Received 10 December 2014; Revised 26 April 2015; Accepted 17 June 2015

KEY WORDS: modeling; global memory transactions; shared memory transactions; thread-block; concurrent thread-blocks; GPU; stencils; convolutions

1. INTRODUCTION

Stencils, also called convolutions in image processing, are very important in scientific, engineering, and emerging applications. A stencil can be defined as a function that updates each point of a regular grid based on the values of its neighbors. The stencil structure remains constant as it moves from one point of the grid to the next. There exists a growing effort in developing new languages, compilers, execution systems, and autotuners for optimizing stencils on the newest high-performance computers [1, 2]. As most stencils are memory-bound kernels, the main strategy to approach their theoretical peak performance on the graphic processing unit (GPU) is by using data tiling through shared memory. This transformation improves locality but introduces synchronization points due to halos and boundary conditions. Halos are the data needed to update the borders of the thread-block.

Since the last decade, GPUs have become the most popular accelerators due to their good ratio price to performance. As a result, the demand of highly optimized kernels for GPUs is also increasing. The most important aspects a programmer considers when optimizing stencil codes for the GPU are: (1) using the appropriate code optimizations and, once having the code; (2) finding the

*Correspondence to: Siham Tabik, Department of Computer Architecture, University of Málaga, 29071, Málaga, Spain.
†E-mail: stabik@uma.es

optimal thread-block configurations. The most common criteria reported in a large number of works are either selecting thread-block sizes that maximize the SM occupancy or using specific sizes and shapes, mainly square shapes, to simplify the programming task and to ease portability among different accelerator architectures [3].

The works that deal with the determination of the best thread-block configurations for GPUs can be classified into three groups. The first group focuses on automating the exhaustive search process [1]. The second category intends to partially reduce the configuration space. For example, [4] calculates the best configuration based only on GPU limitations without considering the particularities of the kernel implementation. The authors showed that their approach is not appropriate for stencils. In [5], Ryoo *et al.* proposed an approach for reducing the optimization space first by defining two metrics to judge the performance of all the configurations and then evaluating the set of a Pareto-optimal configurations. This approach requires the code to be executed on several configurations and does not always guarantee the presence of the best configuration in that set. The third group focuses on understanding the interrelation between the thread-block configuration and different memory access patterns by means of experimental explorations [6–8]. Our own work does not require the code to be executed as the previously cited works and take into account the main characteristics of the (i) GPU, (ii) kernel implementation, and (iii) input size. We build a performance model for stencil computation which can be especially useful for stencils compilers, execution systems and autotuners. The main contributions of this paper can be summarized as follows:

- We accurately model the number of global memory transactions and the number of shared memory transactions for stencil kernels on the GPU.
- We build a performance model that predicts the best thread-block configurations without the need of running the code.
- Provide insights into the interrelation between thread-block size and shape and the performance of stencils.

The rest of the paper is organized as follows. An overview of the GPU architecture is provided in Section 2. A description of the stencils selected for our study is given in Section 3. The performance model that we build for stencil implementations on GPUs is provided in Section 4. The validation of our model is shown in Section 5, and the conclusions are given in Section 6.

2. GPU ARCHITECTURE

Nvidia GPUs are massively parallel processors that provide an advantageous ratio between GFlop/s rate and power consumption. They are characterized by a large number of cores (e.g., 16) called streaming multi-Processors (SM), wide Single Instruction Multiple Data (SIMD) units (e.g., 32 lanes), and a complex memory hierarchy including scratchpad memory (called shared memory) explicitly controllable by the programmer, a two-level cache hierarchy, constant and texture memory, and global memory (random access memory). Because they are in-order processors, GPUs hide pipeline latencies by interleaving the execution of hundreds of warps (e.g., a group of 32 threads) per SM. To support a fast context switch for such a large number of concurrent threads, each SM is equipped with a high number of registers. Threads in Compute Unified Device Architecture (CUDA) are organized into grids, grids are composed of thread-blocks, and each thread-block is executed in warps.

The GPU architecture has known a noticeable evolution in the last 10 years. The main trend in this evolution has been the increase of the number of cores per SM and the number of registers per thread-block as it is shown in Table I. These changes affect directly the configuration of the optimal thread-blocks. In this work, we formulate a model that finds out a set of the best thread-block configurations by taking into account the most important characteristics and limitations of GPU architecture.

3. CASE STUDIES

Stencils are crucial to capture, analyze, mine, and render digital signals such as 2D and 3D images and videos. To carry out our study, we selected six stencils with different data access patterns and

DOI: 10.1002/cpe

Table I. The evolution of NVIDIA GPUs.

Parameter	Tesla	Fermi	Kepler
PE/SM	8	32	192-384
32-bit registers/SM	16384	32768	65536
Maximum number of threads (per SM)	1024	1536	2048
Maximum number of threads (per block)	512	1024	1024
Thread per Warp	32	32	32
Maximum number of blocks/SM	8	8	16
Warp scheduler	Single	Dual	HyperCube
Shared memory (per SM)	4 8KB	48^d/16 KB	48^d/32/16 KB
Number of shmem banks	32	32	32
Bandwidth of each shmem bank	16-32	32	32^d/64
L1 cache (per SM)	—	16^d/48 KB	16^d/32/48 KB
Cache line	128 bytes	128 bytes	128 bytes
L2 cache	—	764 KB	256–1.536 KB
Global memory banks	8	6	4
Size of global memory transaction	32/64/128 B	32/128 B	32/128 B
Year of introduction	Nov. 2006	Mar. 2010	Mar. 2012

^{d} means by default; GPUs, graphics processing units; shmem, shared memory.

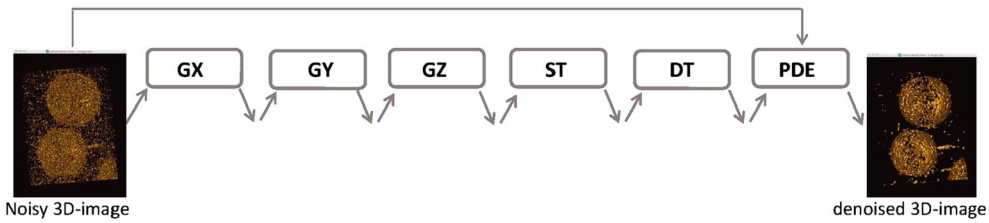


Figure 1. The pipeline, of six-stencils, considered in this work. Upward and downward arrows refer to reads and writes from/to global memory, respectively. Only the first four stages are used in our study.

different arithmetical intensities. Four convolutions from a real-world denoising application in image processing is the anisotropic non-linear diffusion (AND) [9, 10]. AND is a pipeline of six stages as depicted in Figure 1. We only consider the first four stages, GX, GY, GZ, and ST, because PDE is similar to ST and DT is a simple 1-point stencil. Moreover, to cover stencils with larger halos, we consider two individual stencils, 5-FDD and 7-FDD, obtained from finite difference discretization (FDD) of the wave equation in seismic imaging provided in [3]. The used stencils can be described as follows:

1. GX: This stage applies a Gaussian blur in the x-direction. It processes the noisy 3D image of size $N_x \times N_y \times N_z$ and outputs an intermediate 3D image of size $N_x \times N_y \times N_z$. This kernel will be used as example in the next section to illustrate the proposed model.
2. GY: This stencil applies the Gaussian blur in the y-direction. It sweeps the output 3D image of the previous stage and generates a temporal 3D image of the same size.
3. GZ: This stage applies the Gaussian blur in the z-axis dimension. It reads the output of the previous stage and outputs a temporal 3D image.
4. ST: This 3D stencil calculates the structure tensor, which is a 3×3 symmetric matrix, for each input pixel. Only the six elements of the upper part of these 3×3 matrices are stored. ST calculates the structure tensor of the 3D input image and generates a structure of arrays of size $6 \times N_x \times N_y \times N_z$. This stencil is a 7-points 3D stencil.
5. 5-FDD: This 3D stencil processes a 3D array of size $N_x \times N_y \times N_z$ and outputs another 3D array of the same size. This is a 31-points stencil where each pixel needs to read the five nearest neighbors from behind, in front, up, down, right, and left sides. That is, the halo width equals 5 in the three dimensions.

DOI: 10.1002/cpe

6. 7-FDD: This is a 43-points stencil where each pixel needs to read the seven nearest neighbors from behind, in front, up, down, right, and left sides. The halo width equals 7 in the three dimensions.

The input, intermediate, and output 3D images are implemented as a uni-dimensional array.

3.1. Optimizing locality in stencils

This subsection provides an overview of GPU mapping strategies for stencils. For clarity, we use GX stencil as an educational example. The pseudocode of GX is listed in Algorithm 1.

Algorithm 1 Naive implementation

```
for  k=0, k<Nz; k++
 for j=0, j<Ny; j++
   for  i=0, i<Nx; i++
   out[k][j][i]=coef[0]*in[k][j][i+1]+coef[1]*(in[k][j][i]+in[k][j][i+2]);
```

An intuitive mapping for this stencil would be, each thread is in charge of updating one point of the output 3D image by reading all the necessary neighbor points directly from global memory (Figure 2). This strategy has a high degree of parallelism, because each output–point can be updated independently but has a small amount of data-reuse between neighbor points. If we omit the cache effect, this strategy has a total number of accesses to global memory that equals $3 \times N_x \times N_y \times N_z$.

Algorithm 2 Tiling i and j considering one thread

```
for k=0, k<Nz, k++
  for jj=0, jj<Ny, jj+=Blk.y
    for  ii=0, ii<Nx, ii+=Blk.x
      for j=j, j<jj+Blk.y, j++
        for  i=i, i<ii+Blk.x, i++
        out[k][j][i]=coef[0]*in[k][j][i+1]+coef[1]*(in[k][j][i]+in[k][j][i+2])
```

Locality can be increased by tiling the i and j loops. Each input point can be reused three times between three successive neighbor points and transfers to and from global memory can be reduced depending on the values of $Blk.x$ and $Blk.y$. Different values of $Blk.x$ and $Blk.y$ lead to different volumes of memory transfers.

The optimal implementations of 3D stencils on GPUs reported in many studies [1, 3, 11–13] apply a 2D tiling in the x and y dimensions, that is, the i and j loops. No tiling is applied in the Z-dimension. Each thread updates one output pixel. The threads of the thread-blocks traverse the volume along the z-axis and compute one XY plan per iteration. Algorithm 3 shows a pseudocode expressed using CUDA syntax, where tiles are stored in shared memory to take advantage of its

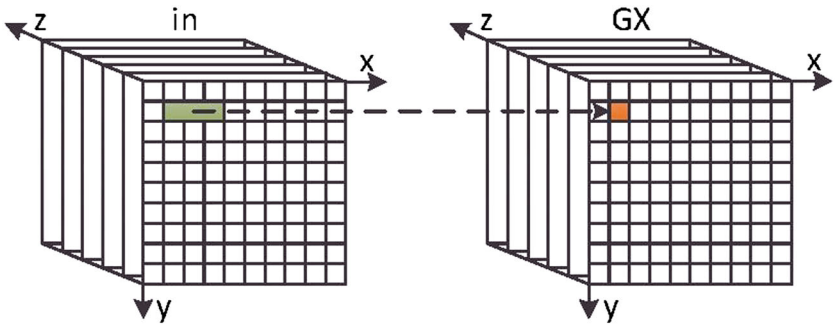


Figure 2. Illustration of data-access pattern in Algorithm 1. Three input points (in green color) are needed to update each output point (in green orange).

DOI: 10.1002/cpe

high bandwidth and low latency. In the algorithm, $Blk.x$ and $Blk.y$ correspond to the width and height of both, the tile and thread-block, with $Blk.x$ =0,...,$N_x - 1$ and $Blk.y = 0,\ldots,N_y - 1$. The scheme of GX data dependencies is plotted in Figure 3(a).

Algorithm 3 Tiling x and y for multi-threads

```
#define  Blk.x tile_X
#define  Blk.y tile_Y

int  k;

int tx= threadIdx.x;
int ty= threadIdx.y;

int gtidx = blockIdx.x*blockDim.x+ tx;

int gtidy = blockIdx.y*blockDim.y+ ty;

__shared float tile[Blk.y][Blk.x+2]

float value, current;
int stride_z   = Nx*Ny;

int inIndex =gtidy*Nx+gtidx;

for(k=0; k<Nz; k++){
  current = in[inIndex];
  __syncthreads();
  if(tx>=Blk.x-2)// read halos
  {
     tile[ty][tx+2]= in[inIndex+2];
  }
  tile[ty][tx]= current;
  __syncthreads();
  value = coef[0] * tile[ty][tx+1] + coef[1]* (tile[ty][tx] + tile[ty][tx+2]);
  out[inIndex]=value;
  inIndex += stride_z;
}
```

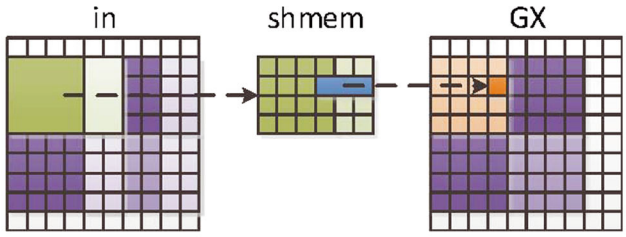
As discussed, tiling has beneficial effects but also has negative ones; it penalizes parallelism by introducing multiple thread-block synchronizations that are necessary for halos. The larger the halo, the higher is the overhead and the higher is the thread divergence.

The GPU implementation of the GY, GZ, ST, 5-FDD, and 7-FDD convolutions have the same structure as the GX implementation, even if their data-dependence patterns are completely different. One of the particularities of the GY stencil is that the allocated space in shared memory is $Blk.x \times (Blk.y+2)$ instead of $(Blk.x+2) \times Blk.y$ as shown in Figure 3(b) (see differences in Table II). As the threads in GZ need to read only neighbor points from different XY plans, that is, only temporal locality is available; it does not make use of shared memory and utilize only registers. In ST, each thread needs to read the neighbors from the right, left, top, bottom, behind, and in front as shown in Figure 3(c). The space allocated in shared memory is $(Blk.x + 2) \times (Blk.y + 2)$.

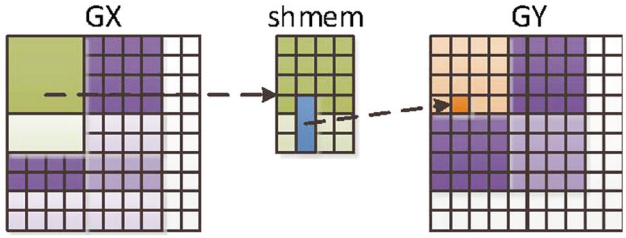
4. OPTIMAL THREAD BLOCK CONFIGURATION

Given a stencil implementation, an input volume of data, and a GPU, we aim at determining the optimal thread-block configurations that provide the best overall performance. The common way

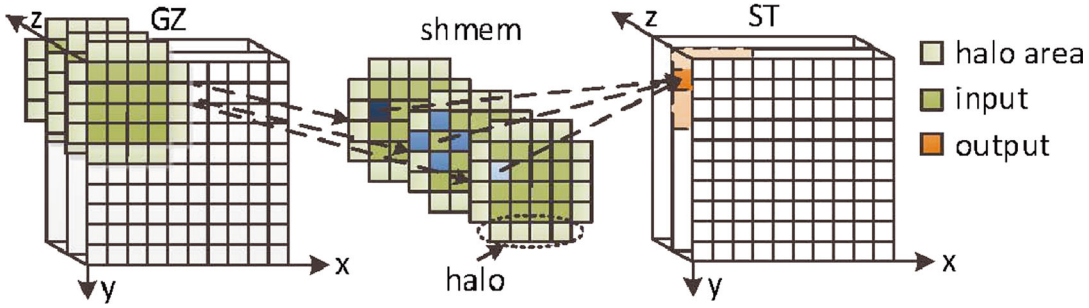
DOI: 10.1002/cpe



(a) In GX, each output-point (in orange color) is updated using three input-points (in blue color).



(b) In GY, each output-point (in orange color) is updated using three input-points (in blue color).



(c) In ST, each output-point (in orange color) is updated using six input-points (in blue color).

Figure 3. A close look into the GX, GY, and ST implementations. shmem visualizes shared memory.

Table II. Shared memory utilization of GX, GY, GZ, ST, 5-FDD, and 7-FDD stencils using the same thread-block configuration $Blk.x \times Blk.y$.

Stencil	shmem usage
GX	$(Blk.x + 2) \times Blk.y$
GY	$Blk.x \times (Blk.y + 2)$
GZ	none
ST	$(Blk.x + 2) \times (Blk.y + 2)$
5-FDD	$(Blk.x + 2 * 5) \times (Blk.y + 2 * 5)$
7-FDD	$(Blk.x + 2 * 7) \times (Blk.y + 2 * 7)$

of finding the optimal configuration is by exhaustively testing the complete space or considering a reduced space of popular square configurations such as 16×16.

The best configurations should achieve a good compromise between locality and parallelism and hide synchronization cost. These configurations are characterized by a reduced number of global memory transactions, a good utilization of shared memory and SMs and high thread-block efficiency. We propose a model that accurately estimates: (1) the number of global memory transactions; (2) number of shared memory transactions; (3) the SM utilization; and (4) thread-block efficiency. Then we consider that the optimal configurations are the ones that obtain the best global memory and shared memory behavior, the highest SM occupancy, and an optimal thread-block

DOI: 10.1002/cpe

Table III. The space of thread-block configurations, $(Blk.x, Blk.y)$, for the GX stencil on 'Geforce GTX TITAN'.

Blk.y \ Blk.x	1	2	4	8	16	32	64	128	256	512
1	X	X	X	X	X					
2	X	X	X	X						
4	X	X	X							
8	X	X								
16	X									
32										
64										
128										
256										
512										

The white cells indicate the valid configurations. The red, light grey, dark grey, and cells with X are impossible configurations or configurations that lead to a sub-optimal utilization of the GPU resources.

efficiency. Our approach first eliminates the configurations that lead to a sub-optimal utilization of the GPU resources and all impossible and illegal configurations due to the physical GPU limitations, stencil-implementation constraints, and input size constraints. Then our performance model is built according to the estimations indicated previously. To make the discussion clearer, we use the GX stencil implementation shown in Algorithm 3 as an example and consider the architecture of 'Geforce GTX TITAN'.

4.1. GPU and implementation limitations

A first pruning of the configuration space is performed based on the GPU, input size, and implementation limitations.

GPU limitations

- To avoid an under-utilization of the GPU resources, thread-blocks must have at least one work-unit, that is, one warp, $Blk.x * Blk.y \geqslant warp_{size}$, where $warp_{size} = 16$ in Fermi and $warp_{size} = 32$ in Kepler. The configurations that do not meet this requirement are indicated by the symbol X in Table III.
- To discard unnecessary divergence and load unbalance, the number of threads per block should be multiple of the warp size, $((Blk.x \times Blk.y) \mod warp_{size}) = 0$, and the thread-block sizes should be divisible by the size of the input.
- The total number of threads per thread-block must be smaller than or equal to a given limit $Threads_{Blk}^{Max}$, $Blk.x * Blk.y \leqslant Threads_{Blk}^{Max}$; invalid thread-block configurations are indicated by the red cells in Table III.
- The data tile including halos must fit in shared memory $(Blk.x + H.x) * (Blk.y + H.y) \leqslant shmem_{size} * sizeof(data)$; $H.x$ and $H.y$ refer to the halo width in the x-direction and y-directions, respectively.
- The used registers per thread-block must be smaller than a given limit, $Reg_{thread} \leqslant Reg_{SM}^{Max}$, for example, 65536 in Kepler.

Input size limitation, $Blk.x < N_x$ and $Blk.y < N_y$. The area in dark gray color in Table III. Implementation limitations, $Blk.x \geqslant H.x$ and $Blk.y \geqslant H.y$. The area in light gray color in Table III.

4.2. Global memory transactions

For memory bound kernels, as it is often the case for stencils, the number of global memory transfers is a key parameter for performance. Reducing global memory transactions in stencils is possible by

DOI: 10.1002/cpe

increasing data reuse in the on-chip shared memory and registers. The amount of data reutilization strongly depends on the shape of the thread block, because it defines the memory portion used in the block. The prediction of the required global memory transactions can be performed based on the general *intra-block communication* model for one thread on a mono-core chip [14]. This model expresses the communication volume of the data loaded into and written out to a block of data of size $Blk.x \times Blk.y$ as follows:

$$\begin{aligned} N_{g.mem.trans} &= N_{Blks} \times (\text{datatransfer/Blk}) \\ &= stores + loads \\ &= \left\lceil \frac{N_x}{Blk.x} \right\rceil \left\lceil \frac{N_y}{Blk.y} \right\rceil \left\lceil \frac{N_z}{Blk.z} \right\rceil \times \\ &\left(\underbrace{Blk.x \times Blk.y \times Blk.z}_{\text{stores}} + \underbrace{(Blk.x + H.x)(Blk.y + H.y)(Blk.z + H_z) + Coeff}_{\text{loads}} \right) \end{aligned} \tag{1}$$

where N_{Blks} is the total number of tiles or data blocks. It is calculated by dividing the total size of the output 3D image by the size of one block of data. N_x, N_y, and N_z are the size of the input 3D image in the x-axis, y-axis, and z-axis. $Blk.x$, $Blk.y$, and $Blk.z$ are the size of the 3D data-block in the x-axis, y-axis, and z-axis. H_x, H_y, and H_z model the halos in the x-dimension, y-dimension, and z-dimension and depend on the GPU implementation of the kernel. $Coeff$ is the number of necessary loads and stores of the stencil weight coefficients, which in general equals zero because the weights are stored in registers, that is, $Coeff = 0$.

Expression (1) must be adapted to the GPU as multiple threads are simultaneously accessing global memory. The number of transactions required per warp request (load or store request) varies depending on the warp access pattern. Next, we analyze typical access patterns in stencil kernels and their impact on the number of global memory transactions.

4.2.1. Global memory transactions per request. To take advantage of coalescence and locality, most stencil studies [1, 3, 11–13, 15] assign consecutive image pixels to consecutive GPU threads. That is, thread-blocks and data blocks have similar sizes $Blk.x \cdot Blk.y$ and consider $Blk.z = 1$ and $H.z = 0$.

As data pixels are stored in row-major order, the number of necessary transactions to load and store one row of the tile depends strongly on the shape and size of the thread-block. For thread-blocks of width larger than or equal to the size of a warp, $Blk.x \geqslant warp_{size}$, the number of required accesses to read one row of data pixels is $\lceil \frac{Blk.x}{warp_{size}} \rceil$ transactions, in addition to $\lceil \frac{H.x}{warp_{size}} \rceil$ transactions per row to load halo pixels. Figure 4(a) illustrates this situation using GX stencil for $warp_{size} = 8$ and $Blk.x = 16$. In this example,three transactions are needed, two for data pixels and one for halo pixels.

For thread-blocks of width smaller than the size of a warp, $Blk.x < warp_{size}$, data accesses are not coalesced as the threads of one warp have to read data located in different rows of the input volume. In this case, the number of non-coalesced accesses needed to read data pixels is $\lceil \frac{warp_{size}}{Blk.x} \rceil$ transactions. This scenario is illustrated in Figure 4(b) using the GX stencil for $Blk.x = 4$ and $warp_{size} = 8$. In this example, four transactions are required, two for data pixels and two for halo pixels.

4.2.2. Global memory transactions in stencil computation. From the analysis shown earlier, the total number of global memory transactions, $N_{g.mem.trans}$, for a given stencil can be reformulated into Expression (2) as follows:

$$\begin{aligned} N_{g.mem.trans} &= \left\lceil \tfrac{N_x}{Blk.x} \right\rceil \left\lceil \tfrac{N_y}{Blk.y} \right\rceil N_z \times \\ &\begin{cases} \left\lceil \frac{Blk.x}{warp_{size}} \right\rceil \times Blk.y + \left\lceil \frac{Blk.x}{warp_{size}} \right\rceil \times (Blk.y + H.y) + \left\lceil \frac{H.x}{warp_{size}} \right\rceil \times Blk.y & Blk.x \geqslant warp_{size} \\ \frac{Blk.x \times Blk.y}{warp_{size}} \times \left[\lceil \frac{warp_{size}}{Blk.x} \rceil + \left\lceil \frac{H.x}{warp_{size}} \right\rceil + \left\lceil \frac{Blk.x}{warp_{size}} \right\rceil \right] + \frac{Blk.x}{warp_{size}} \times H.y & Blk.x < warp_{size} \end{cases} \end{aligned} \tag{2}$$

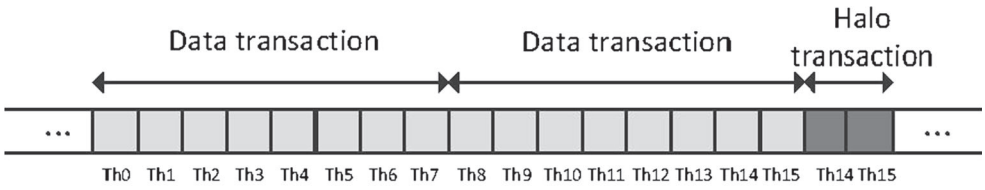
DOI: 10.1002/cpe

To maintain clarity by distinguishing the stores and loads operations, we kept Expression (2) for the cases $Blk.x < warp_{size}$ without applying further mathematical simplifications.

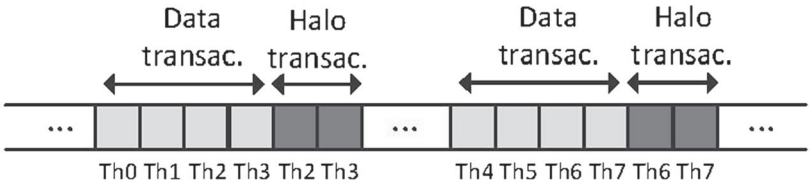
To derive the number of global memory transactions from Expression (2) for GX stencil, we have to substitute $H.y = 0$. Recall that for GX stencil, the halo is needed only in the right side of the tile. Applying some mathematical simplifications for the cases where $Blk.x < warp_{size}$, the total number of global memory transactions for GX for any $Blk.x$ value will be as follows:

$$N_{g.mem.trans} = \left\lceil \frac{N_x}{Blk.x} \right\rceil \left\lceil \frac{N_y}{Blk.y} \right\rceil N_z \times \left(\left(2 \times \left\lceil \frac{Blk.x}{warp_{size}} \right\rceil + \left\lceil \frac{H.x}{warp_{size}} \right\rceil \right) \times Blk.y \right) \quad (3)$$

Figure 5 shows the number of global memory transactions calculated using Formula (3) for GX stencil considering all the valid thread-block sizes and shapes on a 'Geforce GTX TITAN'. For this architecture, $warp_{size} = 32$. As can be observed, the number of required transactions decreases when $Blk.x$ increases. The differences between successive configurations with $Blk.x \geqslant 32$ are slighter than the differences between successive configurations with $Blk.x < 32$. We consider that the optimal configurations are the best 50% configurations, shown in lightest orange cells in Figure 5. All the values shown in Figure 5 are validated using NVIDIA CUDA profiler, nvccprof. This demonstrates that our model for predicting the total number of global memory transactionssss is exact.



(a) Required transactions per row(of the data-block) for $warp_{size} = 8$, $H.x = 2$ and $Blk.x = 16$.



(b) Required transactions per warp for $warp_{size} = 8$, $H.x = 2$ and $Blk.x = 4$.

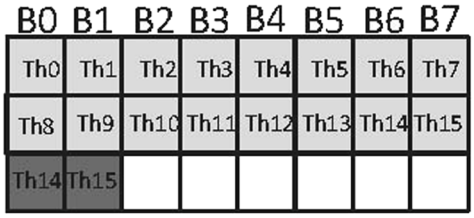
Figure 4. The global memory transactions required for the GX stencil considering $warp_{size} = 8$. For $Blk.x = 16$ (a) and $Blk.x = 4$ (b).

Global memory transactions

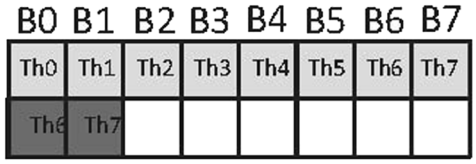
Blk.y\Blk.x	1	2	4	8	16	32	64	128	256	512
1	X	X	X	X	1572864	1310720	1179648	1114112		
2	X	X	X	3145728	1572864	1310720	1179648	1114112		
4	X	X	6291456	3145728	1572864	1310720	1179648	1114112		
8	X	12582912	6291456	3145728	1572864	1310720	1179648			
16	25165824	12582912	6291456	3145728	1572864	1310720				
32	25165824	12582912	6291456	3145728	1572864					
64	25165824	12582912	6291456	3145728						
128	25165824	12582912	6291456							
256	25165824	12582912								
512										

Figure 5. The number of global memory transactions for each configuration ($Blk.x$, $Blk.y$) for the GX convolution. The light orange cells highlight the 50% best configurations.

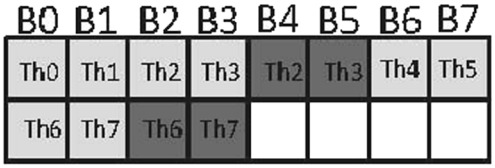
DOI: 10.1002/cpe



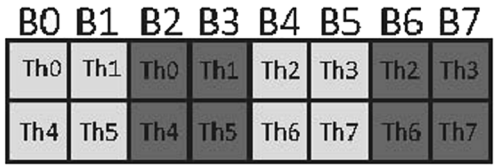
(a) Required transactions per row for $warp_{size} = 8$, $H.x = 2$ and $Blk.x = 16$.



(b) Required transactions per row for $warp_{size} = 8$, $H.x = 2$ and $Blk.x = 8$.



(c) Required transactions per row for $warp_{size} = 8$, $H.x = 2$ and $Blk.x = 4$.



(d) Required transactions per row for $warp_{size} = 8$, $H.x = 2$ and $Blk.x = 2$.

Figure 6. Shared memory transactions for different $Blk.x$ values.

4.3. Shared memory transactions

The number of shared memory transactions is also critical to the performance of memory-bound kernels. Current GPUs have a shared memory with a number of banks similar to the warp size, allowing free conflict access patterns. However, stencil access patterns can generate bank conflicts, which increase the expected number of memory transactions. This subsection describes a model that accurately predicts the number of shared memory transactions for stencil kernels.

4.3.1. Shared memory transactions per request. Load and store instructions on shared memory can suffer from bank conflicts. These conflicts increase the shared memory latency, because the memory warp requests have to wait till all the required accesses are finished. The bank conflict degree of a warp is the maximum number of memory requests that fall in the same bank. According to [16], bank conflicts can have a high impact on shared memory performance as memory latency increases linearly with respect to the bank conflict degree.

For store operations, no bank conflicts occur when $Blk.x \geqslant warp_{size}$. This scenario is illustrated in Figures 6(a) and (b) using GX stencil, $warp_{size} = 8$, considering a shared memory of eight banks for $Blk.x = 16$ and $Blk.x = 8$, respectively. The number of memory accesses required to read one row of the tile is given by $\left\lceil \frac{Blk.x}{warp_{size}} \right\rceil$ in addition to $\left\lceil \frac{H.x}{warp_{size}} \right\rceil$ transactions to read the halo of one row. However, bank conflicts appear when $Blk.x < warp_{size}$. Figure 6(c) illustrates an example with $Blk.x = 4$, where the memory addressing generates a two-degree bank conflict when data pixels are stored. Notice that in this case there is no bank conflicts for halo pixels stores. Therefore, the number of transactions per warp is three in this particular case. Figures 6(d) shows another possible scenario, with $Blk.x = 2$. In this case, a two-degree bank conflict occurs for both data-pixels and halo-pixels writes, requiring four transactions.

For load operations, no bank conflicts occur when reading a row of the tile for $Blk.x \geqslant warp_{size}$. So, the number of needed transactions can be given by $\left\lceil \frac{Blk.x}{warp_{size}} \right\rceil$ transactions per block row. Whereas, when $Blk.x = 8$, two transactions are required per warp as no halo pixels are read. For $Blk.x = 4$, also two transactions are needed per warp.

4.3.2. Shared memory transactions in stencil computation. From previous analysis, we compute the number of shared memory transactions required for a specific stencil in GPU. Without loss of generality, we show in this section how the total number of store and load transactions can be deduced for GX stencil:

$$N_{shmem.trans} = N_{stores.trans} + N_{loads.trans} \tag{4}$$

DOI: 10.1002/cpe

Shared memory transactions

Blk.y\Blk.x	1	2	4	8	16	32	64	128	256	512
1	X	X	X	X	2621440	2359296	2228224	2162688		
2	X	X	X	4718592	2621440	2359296	2228224	2162688		
4	X	X	4718592	4718592	2621440	2359296	2228224	2162688		
8	X	4718592	4718592	4718592	2621440	2359296	2228224			
16	5242880	4718592	4718592	4718592	2621440	2359296				
32	5242880	4718592	4718592	4718592	2621440					
64	5242880	4718592	4718592	4718592						
128	5242880	4718592	4718592							
256	5242880	4718592								
512										

Figure 7. The total number of shared memory transactions calculated using our shared memory transactions model for the example of GX stencil and considering the characteristics of 'Geforce GTX TITAN'.

$$N_{stores.trans} = \left(F_{BCD.stores} \times \left\lceil \frac{Blk.x}{warp_{size}} \right\rceil + F_{BCH.stores} \times \left\lceil \frac{Hx}{warp_{size}} \right\rceil \right) (Blk.y + Hy) \left\lceil \frac{N_x}{Blk.x} \right\rceil \left\lceil \frac{N_y}{Blk.y} \right\rceil N_z \tag{5}$$

where $F_{BCD.stores}$ and $F_{BCH.stores}$ represent the bank conflict degree for data and halo, respectively. These two factors are defined as

$$F_{BCD.stores} = \begin{cases} 1 & Blk.x \geqslant warp_{size} \\ 2 & Blk.x < warp_{size} \end{cases} \tag{6}$$

$$F_{BCH.stores} = \begin{cases} 1 & Blk.x \geqslant warp_{size} \\ 2 & Blk.x < warp_{size} \end{cases} \tag{7}$$

$$N_{loads.trans} = num_{reads} \times \left(\left\lceil \frac{Blk.x}{warp_{size}} \right\rceil + F_{BCH.loads} \right) (Blk.y + Hy) \left\lceil \frac{N_x}{Blk.x} \right\rceil \left\lceil \frac{N_y}{Blk.y} \right\rceil N_z \tag{8}$$

where $F_{BCH.loads}$ is the degree of bank conflicts when reading halos. num_{reads} is the number of reads required to update one single pixel. This factor can take the next values:

$$F_{BCH.loads} = \begin{cases} 0 & Blk.x \geqslant warp_{size} \\ 2 & Blk.x < warp_{size} \end{cases} \tag{9}$$

Figure 7 depicts the number of shared memory transactions calculated using our model for GX kernel. As can be observed from this figure, the number of transactions increase with $Blk.x$. The thread-blocks with width $Blk.x \geqslant 32$ show better numbers than the cases with $Blk.x < 32$. Again, we select the 50% best configurations as candidates, shown in light orange cells.

4.4. SM utilization

Different shapes and sizes of the thread-block also lead to different utilization of the SMs. To express the SM utilization, we first need to define the number of active blocks per SM, $Blks_{SM}^{active}$. The $Blks_{SM}^{active}$ depends on the maximum number of registers and shared memory space available per SM and also on the maximum block size allowed per SM, which are specific to each GPU architecture.

$$Blks_{SM}^{active} = min \left(\left\lfloor \frac{Regs_{SM}^{Max}}{Regs_{Blk}^{stencil}} \right\rfloor, \left\lfloor \frac{shmem_{SM}^{Max}}{shmem_{Blk}^{stencil}} \right\rfloor, Blk_{SM}^{Max}, \frac{Threads_{SM}^{Max}}{Blk.x \times Blk.y} \right) \tag{10}$$

where $Blk_{SM}^{Max} = 8$ for Fermi and $Blk_{SM}^{Max} = 16$ for Kepler. The amount of shared memory used by block is determined from the kernel implementation, which can be formulated in general as

DOI: 10.1002/cpe

$shmem_{Blk}^{stencil} = (Blk.x + H.x) \times (Blk.y + H.y) \times sizeof(data)$ bytes. $Threads_{SM}^{Max} = 1024$ in Fermi and $Threads_{SM}^{Max} = 2048$ in Kepler and Maxwell families. $Threads_{block}^{Max} = 1024$ in Fermi, Kepler, and Maxwell. $Regs_{Blk}^{stencil}$ is provided by the nvcc compiler after compilation.

$$Occupancy_{SM}^{stencil} = \frac{Blks_{SM}^{active} \times Warps_{Blk}^{stencil}}{Warps_{SM}^{Max}} \tag{11}$$

$Warps_{SM}^{Max} = 64$ in Kepler. The configurations that provide the optimal SM utilization, 100%, are shown in light orange in Figure 8.

4.5. Thread-block efficiency

The size and shape of the thread-block affect the cost of synchronizations and, consequently, the efficiency of threads within a thread-block. Stencil implementations include synchronizations mainly to allow a number of the threads to read halos while the remaining threads are idle. The higher the number of idle threads within the thread-block, the less efficient they are. Consider the example of GX stencil, in each thread-block, $H.x \times Blk.y$ threads have to read the halo while $Blk.x \times Blk.y - H.x \times Blk.y$ are idle. Notice that storing in shared memory halos from the x-axis is more expensive than storing halos from the y-axis, because they produce more conflicts. Indeed, the performance analysis provided in the next section for stencils GX, GY, and ST validate this statement. Stencil GZ does not have thread-efficiency issues because no synchronization is needed due to the nature of its data access pattern.

The thread-block efficiency can be expressed as

$$Blk_{Eff} = \frac{Time_{work}}{Time_{work} + Time_{idle}} \tag{12}$$

where $Time_{work}$ refers to the time spent only in doing useful work, that is, computing and, reading and writing from/to memory. $Time_{idle}$ is the idle time waiting for halos to be stored in shared memory. As one can observe from this formula, the larger is the idle time, the less efficient is the

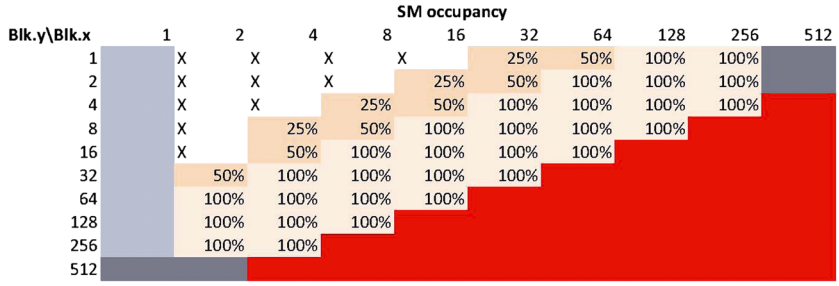


Figure 8. The occupancy of the SM for all the possible configuration space. The lighter orange indicates the best 50% configurations.

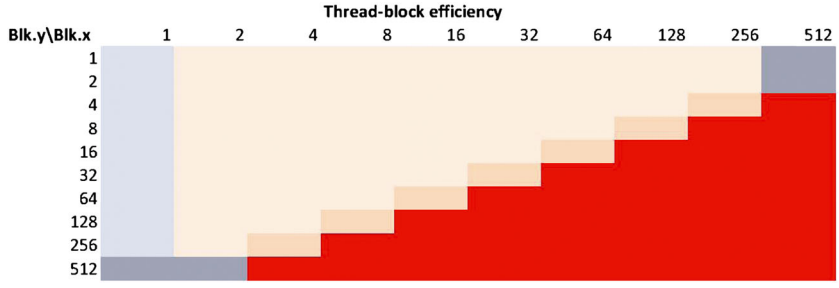


Figure 9. The light orange cells show the configurations that produce the smallest number of active thread-blocks per SM, which is 2 for the GX stencil.

DOI: 10.1002/cpe

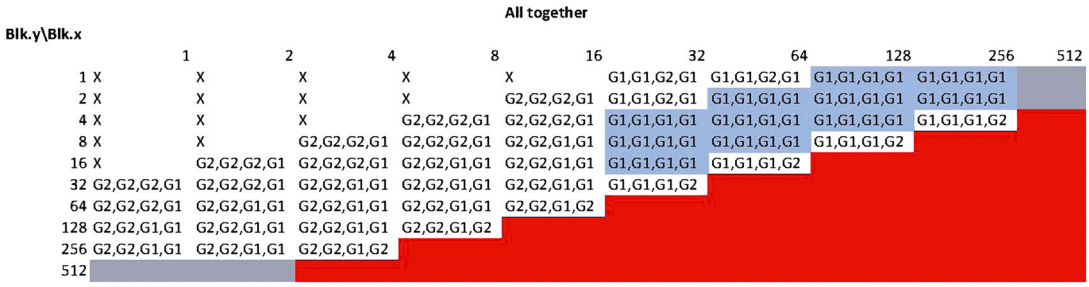


Figure 10. G1 and G2 stand for the best and worst group of configurations in each aspect, global memory behavior, shared memory behavior, SM occupancy, and thread-block efficiency. The configurations that provide the best behavior in all aspects are indicated by G1,G1,G1,G1 in blue color.

Table IV. Execution times (in milliseconds) on all the valid thread-block configurations $(Blk.x, Blk.y)$ for GX stencil using an input 3D image of size 256 × 256 × 256 on 'Geforce GTX TITAN'.

Blk.y \ Blk.x	1	2	4	8	16	32	64	128	256	512
1		X	X	X	X	1.60	0.97	0.75*	0.77*	
2		X	X	X	1.92	1.01	0.77*	0.79	0.83	
4		X	X	2.34	1.34	0.81	0.81	0.86	0.95	
8		X	2.83	1.75	1.23	0.86	0.87	0.97		
16		1.92	2.88	1.83	1.24	0.90	1.01			
32		4.80	2.91	1.71	1.24	1.02				
64		5.22	2.99	1.78	1.38					
128		5.15	2.93	1.86						
256		5.17	3.04							
512										

Table V. The execution time (in milliseconds) on all the valid thread-block space using GX stencil for an input 3D image of size 256 × 256 × 256, on 'NVIDIA GeForce GTX 480'.

Blk.y \ Blk.x	1	2	4	8	16	32	64	128	256	512
1		X	X	X	X	3.06	1.73	1.16	0.98*	
2		X	X	X	3.37	1.78	1.17	0.99*	1.01*	
4		X	X	3.89	1.99	1.22	1.03*	1.03*	1.74	
8		X	4.98	2.37	1.64	1.15	1.09	1.91		
16		9.22	5.05	3.32	2.05	1.22	1.98			
32		9.58	5.36	3.01	1.77	2.00				
64		12.85	5.84	2.86	2.19					
128		14.92	5.65	3.19						
256		13.73	5.77							
512										

thread-block. That is, the larger is $Blk.x \times Blk.y - H.x * Blk.y$, the less efficient is the thread-block. The main strategy to hide thread-blocks inefficiency is by increasing the number of simultaneous running blocks in the same SM. This allows the overlapping between concurrent thread-blocks and consequently increases SM resources utilization. The best configurations in this case can be represented by the ones that allow a higher number of active thread-blocks. We conservatively consider that the best configurations are the ones that allows an active thread-blocks per SM higher than the minimum possible, as shown in Figure 9.

DOI: 10.1002/cpe

Table VI. The execution time (in milliseconds) on all the valid thread-block configurations for GX stencil using an input 3D image of size 512 × 512 × 512, on 'NVIDIA GeForce GTX 480'.

Blk.y \ Blk.x	1	2	4	8	16	32	64	128	256	512
1		X	X	X	X	23.65	13.24	8.74	7.95*	7.87*
2		X	X	X	27.51	14.37	9.03	8.29	8.04*	12.73
4		X	X	32.26	17.21	10.63	9.08	8.51	13.58	
8		X	45.37	26.00	16.85	11.13	9.49	14.30		
16		86.14	43.25	24.52	14.93	10.45	14.25			
32		95.69	48.34	25.69	14.70	15.16				
64		124.60	55.81	25.19	17.23					
128		135.42	53.15	26.73						
256		137.71	49.99							
512		89.20								

Table VII. Time (ms) on different thread-block configurations for GY filter using an input size 256 × 256 × 256 on 'Geforce GTX TITAN'.

Blk.y \ Blk.x	1	2	4	8	16	32	64	128	256	512
1										
2	X	X	X	X	X	0.96	0.80*	0.85	0.86	
4	X	X	X	2.15	1.15	0.79*	0.82*	0.85	0.97	
8	X	X	2.45	1.58	1.07	0.82*	0.85	0.99		
16	X	3.38	2.05	1.60	1.00	0.84	0.98			
32	5.88	3.45	1.88	1.31	0.96	0.99				
64	6.02	3.38	1.85	1.27	1.07					
128	6.80	3.51	1.94	1.37						
256	7.17	3.44	2.02							
512										

Table VIII. Time (ms) on all the valid thread-block configurations for GZ filter using an input size 256 × 256 × 256 on 'Geforce GTX TITAN'.

Blk.y \ Blk.x	1	2	4	8	16	32	64	128	256	512
1	X	X	X	X	X	1.29	0.83	0.68*	0.67*	
2	X	X	X	X	1.48	0.81	0.66*	0.67	0.66*	
4	X	X	X	1.79	0.93	0.66*	0.66*	0.67*	0.67	
8	X	X	2.28	1.31	0.74	0.66*	0.66*	0.68*		
16	X	2.88	1.49	1.29	0.72	0.66*	0.67*			
32	5.24	2.80	2.18	1.07	0.71	0.68*				
64	6.15	4.32	1.89	1.04	0.72					
128	8.65	3.95	1.69	1.01						
256	8.84	4.13	1.70							
512										

4.6. The optimal configuration set for GX

As we stated before, we consider that the optimal thread-block configurations are the ones that belong to the optimal sets in global memory behavior, shared memory behavior, SM utilizations, and good efficiency. Figure 10 depicts this set for the GX kernel and considers the 'Geforce GTX TITAN' limitations. The candidates for optimal configurations are shown by the blue colored cells.

DOI: 10.1002/cpe

Table IX. Time (ms) on all the possible thread-block configurations for ST kernel using an input size 256 × 256 × 256 on 'Geforce GTX TITAN'.

Blk.y \ Blk.x	1	2	4	8	16	32	64	128	256	512
1	X	X	X	X	X	2.62	2.60	2.59	2.52	
2	X	X	X	X	3.64	2.63	2.53	2.45*	2.40*	
4	X	X	X	6.90	6.90	2.57	2.58	2.43*	2.92	
8	X	X	9.21	8.23	4.39	2.54	2.46*	2.91		
16	X	19.11	14.24	8.40	4.30	2.64	2.92			
32	31.45	23.34	14.14	7.19	3.50	3.06				
64	41.56	26.53	13.19	6.42	3.62					
128	53.28	27.35	12.70	6.03						
256	52.42	25.93	12.18							
512										

Table X. Time (ms) on all the possible thread-block configurations for 5-FDD using an input size 256 × 256 × 256 on 'Geforce GTX TITAN'.

Blk.y \ Blk.x	1	2	4	8	16	32	64	128	256	512
1	X	X	X	X	X	X	X	X	X	
2	X	X	X	X	X	X	X	X	X	
4	X	X	X	X	X	X	X	X	X	
8	X	X	X	5.25	3.72	2.40	2.80	2.98		
16	X	X	X	4.71	3.66	2.78	3.11			
32	X	X	X	4.58	4.03	3.08				
64	X	X	X	4.96	4.37					
128	X	X	X	5.28						
256	X	X	X							
512										

Table XI. Time (ms) on all the possible thread-block configurations for 7-FDD using an input size 256 × 256 × 256 on 'Geforce GTX TITAN'.

Blk.y \ Blk.x	1	2	4	8	16	32	64	128	256	512
1	X	X	X	X	X	X	X	X	X	
2	X	X	X	X	X	X	X	X	X	
4	X	X	X	X	X	X	X	X	X	
8	X	X	X	6.19	4.44	2.93	3.41	3.65		
16	X	X	X	5.78	4.29	3.36	3.67			
32	X	X	X	5.60	4.88	3.69				
64	X	X	X	6.03	5.23					
128	X	X	X	6.36						
256	X	X	X							
512										

5. EXPERIMENTS

For the purpose of validating our model, we carried out two sets of experiments. The first experiments analyze the execution times using all valid configurations for GX stencil on two GPU generations, 'GeForce GTX 480' and 'Geforce GTX TITAN', and using two input sizes, 3D images of sizes 256 × 256 × 256 and 512 × 512 × 512. The second experiments analyze the five remaining stencils, GY, GZ, ST, 5 × 5 × 5 stencil, and 7 × 7 × 7 stencil on all the possible configuration space using an input of size 256 × 256 × 256 on 'Geforce GTX TITAN'. Then, we compare the best

DOI: 10.1002/cpe

configurations based on the execution times with the best configurations predicted by our model. We used CUDA compiler version 4.0 on 'GeForce GTX 480' and version 6.5 on 'Geforce GTX TITAN'.

Tables IV, V, VI, VII, VIII, IX, X and XI show the average execution time of 10 executions of the GX, GY, GZ, ST, 5-FDD, and 7-FDD for all possible configurations. The cells in blue color show the optimal space predicted by our model. The values with the symbol * indicates the configurations with the best execution time. We consider that the configurations with runtimes around 5% larger than the optimum one belong to the optimal configurations set. As can be observed, all the optimal configurations predicted by our model have lower execution time. Most importantly, the best configurations, that is, the configurations that provide the best execution times always belong to the space predicted by our model. For example, comparing Table IV and Table V, one can observe that the best configurations for the same stencil and same input size are (128,1), (256,1), (64,2) on 'GTX TITAN' and (256,1), (256,2), (128,2), (128,4), (64,4) on 'GTX 480'.

From Table VIII, the best 15 thread-block configurations give competitive results for GZ. As the GPU implementation of GZ does not include any synchronization, all the configurations that provide the best occupancy have a good thread-block efficiency.

Regarding the impact of the halo width on the performance of 3D stencils, Tables X and XI demonstrate that the configurations with Blk.x= multiple of $warp\_size$ keep being the best configurations because they guarantee lower conflicts in shared memory and a reduced number of global memory accesses. Recall that stencil-based codes are well known to be memory bound, therefore the use of configurations that improve the memory management is crucial for the performance independently of the width of the halo.

6. CONCLUSIONS

In this work, we built a model that determines the best set of thread-block configurations for stencils on the GPU. The best configuration is **not unique**, multiple thread-block shapes and sizes can lead to competitive performance. The model takes into account the global memory and shared memory transactions, the SM utilization, and thread-block efficiency. We validated the proposed model using six stencils of different memory access patterns, halo width, and input size on different GPUs. We were able to reduce the space to 25% without missing the configurations with the highest performance. Moreover, we provided insights into the impact of the thread-block size and shape on the global memory, shared memory behavior, and SMs utilization. As future works, we are working on extending the proposed performance model to determine whether or not fusing multiple successive stencils of a given pipeline can be good for performance.

ACKNOWLEDGEMENTS

This work was partially supported by the Spanish Ministry of Science through project TIN 2013 42253-P, the Junta de Andalucia through projects P11-TIC7176 and TIC-8260, and the Dutch Technology Foundation STW through project NEST 10346.

REFERENCES

1. Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 2013; **48**(6): 519–530.
2. Holewinski J, Pouchet L-N, Sadayappan P. High-performance code generation for stencil computations on GPU architectures. *Proceedings of the 26th acm international conference on supercomputing,* ACM, San Servolo Island, Venice, Italy, 2012; 311–320.
3. Micikevicius P. 3D finite difference computation on GPUs using CUDA. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, Washington D.C., US, 2009.
4. White A, Lee S-Y. Derivation of optimal input parameters for minimizing execution time of matrix-based computations on a {GPU}. *Parallel Computing* 2014; **40**(10):628–645.
5. Ryoo S, Rodrigues CI, Stone SS, Baghsorkhi SS, Ueng S-Z, Stratton JA, Hwu W-mW. Program optimization space pruning for a multithreaded GPU. *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization,* ACM, Boston, Massachusetts, US, 2008; 195–204.

DOI: 10.1002/cpe

6. Torres Y, Gonzalez-Escribano A, Llanos DR. Understanding the impact of CUDA tuning techniques for Fermi. *2011 International Conference on High Performance Computing and Simulation (HPCS)*, Istanbul, Turkey, 2011; 631–639.
7. Torres Y, Gonzalez-Escribano A, Llanos DR. uBench: exposing the impact of CUDA block geometry in terms of performance. *The Journal of Supercomputing* 2013; **65**(3):1150–1163.
8. Zhang Y, Owens JD. A quantitative performance analysis model for GPU architectures. *2011 IEEE 17th International Symposium on High Performance ComputerArchitecture (HPCA)*, IEEE, San Francisco Bay Area, California, USA, 2011; 382–393.
9. Fernandez JJ, Li S. Anisotropic nonlinear filtering of cellular structures in cryo-electron tomography. *Computing in Science & Engineering* 2005; **7**(5):54–61.
10. Tabik S, Garzón EM, García I, Fernández JJ. High performance noise reduction for biomedical multidimensional data. *Digital Signal Processing* 2007; **17**(4):724–736.
11. Tabik S, Murarasu A, Romero LF. Evaluating the fission/fusion transformation of an iterative multiple 3D-stencil on GPUs. *1st International Workshop on High-Performance Stencil Computations (HiStencils 2014)*, Vienna, Austria, 2014; 81–88.
12. Zhao Y. Lattice Boltzmann based PDE solver on the GPU. *The Visual Computer* 2008; **24**(5):323–333.
13. Podlozhnyuk V. Image convolution with CUDA. *NVIDIA Corporation White Paper* 2007; **2097**(3), Dresden, Germany.
14. Peemen M, Mesman B, Corporaal H. Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators. *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2015*, 2015.
15. Tabik S, Murarasu A, Romero LF. Anisotropic nonlinear diffusion for filtering 3D images on GPUs. *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, Madrid, Spain, 2014; 339–345.
16. Gomez-Luna J, Gonzalez-Linares JM, Benavides Benitez JI, Guil Mata N. Performance modeling of atomic additions on GPU scratchpad memory. *IEEE Transactions on Parallel and Distributed Systems* 2013; **24**(11):2273–2282.

DOI: 10.1002/cpe