J Supercomput
DOI 10.1007/s11227-015-1392-1

An analytical GPU performance model for 3D stencil computations from the angle of data traffic

Huayou Su · Xing Cai · Mei Wen · Chunyuan Zhang

Abstract The achievable GPU performance of many scientific computations is not determined by a GPU's peak floating-point rate, but rather how fast data are moved through different stages of the entire memory hierarchy. We take low-order 3D stencil computations as a representative class to study the reachable GPU performance from the angle of data traffic. Specifically, we propose a simple analytical model to estimate the execution time based on quantifying the data traffic volume at three stages: (1) between registers and on-streaming multiprocessor (SMX) storage, (2) between on-SMX storage and L2 cache, (3) between L2 cache and GPU's device memory. Three associated granularities are used: a CUDA thread, a thread block, and a set of simultaneously active thread blocks. For four chosen 3D stencil computations, NVIDIA's profiling tools are used to verify the accuracy of the quantified data traffic volumes, by examining a large number of executions with different problem sizes and thread block configurations. Moreover, by introducing an imbalance coefficient, together with the known realistic memory bandwidths, we can predict the execution time usage based on the quantified data traffic volumes. For the four 3D stencils, the average error of

H. Su (✉) · M. Wen · C. Zhang
School of Computer, National University of Defense Technology, Changsha, China
e-mail: huayousu@163.com; shyou@nudt.edu.cn

M. Wen
e-mail: meiwen@nudt.edu.cn

C. Zhang
e-mail: cyzhang@nudt.edu.cn

X. Cai
Simula Research Laboratory, Oslo, Norway
e-mail: xingca@simula.no

X. Cai
Department of Informatics, University of Oslo, Oslo, Norway

Published online: 26 February 2015

the time predictions is 6.9 % for a baseline implementation approach, whereas for a blocking implementation approach the average prediction error is 9.5 %.

Keywords Analytical performance modeling · GPU · Stencil computation · Data traffic

1 Introduction

On modern computing hardware, the disparity between the floating-point capability and memory bandwidth can pose a serious performance limitation. Let us take for example the widespread Tesla K20 GPU [16] from NVIDIA. It has 1.17 Tflop/s as the peak double-precision rate, whereas the peak bandwidth to the device memory is only 208 GB/s. In other words, to reach the peak floating-point performance, every double-precision value (8 bytes) that is loaded/stored from/to the device memory should sustain 45 floating-point operations. Such an arithmetic intensity is impossible to achieve for most types of scientific computation. The achievable GPU performance of these computations is thus bound by the amount of data traffic, instead of floating-point operations.

Stencil-based computations, which are widely used in many branches of computational science, are well known to have their performance determined by data traffic, especially for low-order stencils. Such computations have an overall algorithmic form of repeatedly sweeping through a structured computational mesh, where calculation at each mesh point depends on a fixed number of neighboring mesh points, i.e., a computational stencil. Due to the simple data structures and regular accesses involved, stencil computations are good candidates for porting to the powerful GPU architecture [14], even though it is impossible to achieve the theoretical peak floating-point performance. Exploiting data locality in on-chip storage and adjusting thread granularity are two important approaches to achieving high performance on GPUs. Spatial blocking (also termed chunking) [13] is a commonly used optimization method to enhance the GPU performance of stencil computations. The achievable performance improvement is, however, highly sensitive to the choice of various blocking parameters and the thread block configuration. Blind performance tuning can thus be very time-consuming to identify the optimal chunking size and thread block configuration, due to a large search space. On the other hand, an insightful understanding of the achievable GPU performance will be very helpful.

To understand the realistically reachable GPU performance of stencil computations, while also pinpointing performance bottlenecks, we propose in this paper a simple analytical model from the angle of data traffic. More specifically, we focus on quantifying the volume of data traffic through three stages of the entire GPU memory hierarchy: (1) between registers and the storage on the streaming multiprocessors (SMXs), (2) between the on-SMX storage and the L2 cache, (3) between the L2 cache and GPU's device memory. A set of general formulas is derived for low-order 3D stencil computations with help of three granularities: a CUDA thread, a thread block, and a set of simultaneously active thread blocks. Thereafter, using realistic bandwidths measured by simple benchmarks, we can pinpoint the exact bottleneck of data traffic

and thereby estimate the execution time usage of 3D stencil computations on modern GPUs. Experimental results with four stencils on NVIDIA's Kepler K20 GPU show that the accuracy of the three quantified data-traffic volumes is very high. The average error of the time usage predictions is 6.9 % for baseline implementations and 9.5 % for implementations based on blocking in the z-direction.

The main contributions of this paper are as follows:

1. We derive a set of detailed yet general formulas to quantify the data traffic volumes of low-order 3D stencil computations through the three stages mentioned above.
2. We propose an analytical model to estimate the time usage of 3D stencil computations on a GPU, by combining the quantified data traffic volumes and realistic memory bandwidths. In addition, we introduce an imbalance coefficient to fine-tune the estimated time usage.

The organization of the remainder of the paper is as follows: Sect. 2 provides the background of GPU and CUDA programming related to stencil computations. We detail the modeling methodology in Sect. 3, whereas Sect. 4 presents an experimental evaluation. The related work is reviewed in Sect. 5, and a summary of our findings is given in Sect. 6.

2 GPU architecture and CUDA implementation of stencil computations

2.1 NVIDIA's Kepler architecture

The hardware architectures of NVIDIA's GPUs are highly similar with each other, due to sharing the same programming model named CUDA [17]. In this paper, without loss of generality, we will only consider the Kepler architecture from NVIDIA. A GPU consists of several *streaming multiprocessors*, abbreviated as SMXs for the Kepler architecture [16]. From the aspect of programming model, a CUDA program consists of two parts: a host program and one or several CUDA kernels, which are configured as 1/2/3D thread blocks. Consecutive 32 threads in the same block are grouped as a warp, executed in a SIMD pattern on the same SMX. Threads from different thread blocks can only communicate with each other through the global device memory.

The entire memory hierarchy of a NVIDIA's GPU has four levels. On the top, there is an off-chip global memory for the entire GPU device. At the second level, an on-chip L2 cache is shared across all SMXs, with the purpose of caching data loaded from the global memory. This means that the simultaneously active thread blocks may share data through the L2 cache. At the third level, each SMX has its own storage including L1 cache, shared memory, and read-only data cache on the Kepler architecture. The on-SMX storage is shared among the threads per SMX. It implies that thread blocks allocated to different SMXs will not share data and can be considered independent on this level. At the bottom, each thread has access to a number of private registers. We can thus think of three connections within the entire memory hierarchy: global memory $\leftrightarrow$ L2 cache $\leftrightarrow$ on-SMX storage $\leftrightarrow$ registers. The bandwidths of the three connections are considerably different, with the bandwidth of global memory being the lowest. The volumes of data traffic may also differ considerably, with the largest one being that between the on-SMX storage and registers.

2.2 Baseline CUDA implementation of stencil computations

A very simple example of low-order 3D stencil computation is as follows:

$$u^{\text{new}}_{i,j,k} = \alpha u^{\text{old}}_{i,j,k} + \beta \left(u^{\text{old}}_{i-1,j,k} + u^{\text{old}}_{i+1,j,k} + u^{\text{old}}_{i,j-1,k} + u^{\text{old}}_{i,j+1,k} + u^{\text{old}}_{i,j,k-1} + u^{\text{old}}_{i,j,k+1} \right)$$
$$1 \leq (i, j, k) \leq (N_x, N_y, N_z), \tag{1}$$

where α and β are scalar constants, N_x, N_y and N_z denote the number of computational points in the three spatial directions. It can be seen that the above computation uses a 7-point stencil, in that computing $u^{\text{new}}_{i,j,k}$ requires seven point values of u^{old}. This stencil computation, to be denoted as 7PT-1 throughout the following text, may arise from solving the Laplace equation using finite difference discretization and the Jacobi iterative solver.

The most important content of a straightforward CUDA implementation of the 7PT-1 stencil is as follows:

```
dim3 grid(Nx, Ny, Nz);
dim3 block(Bx, By, Bz);
stencil<<<grid, block>>> (u_old, u_new, alpha,
                          beta, pitch, Nx, Ny, Nz);

 __global__ void stencil(double *u_old, double *u_new,
                         double alpha, double beta,
                         int pitch, int Nx, int Ny, int Nz)
{
  int stride_z = pitch*(Ny+2);
  int gidx = blockIdx.x*blockDim.x + threadIdx.x + 1;
  int gidy = blockIdx.y*blockDim.y + threadIdx.y + 1;
  int gidz = blockIdx.z*blockDim.z + threadIdx.z + 1;
  int gid  = gidz*stride_z + gidy*pitch + gidx;
  if (0<gidx<=Nx && 0<gidy<=Ny && 0<gidz<=Nz)
    u_new[gid] = alpha*u_old[gid]
               + beta*(u_old[gid-1] + u_old[gid+1]
               + u_old[gid-pitch] + u_old[gid+pitch]
               + u_old[gid-stride_z] + u_old[gid+stride_z]);
}
```

In the above code segment, `pitch` is a variable that stores the number of values allocated in the x-direction including padding, similar to the standard CUDA function `cudaMallocPitch()` [17]. As can be seen in the kernel function `stencil`, each CUDA thread is responsible for computing u^{new} at a single mesh point. Throughout the remaining text, we will label one-thread one-point CUDA implementations as *baseline*. Improvement of such straight forward implementations is possible. For example, the technique of blocking lets each thread compute a column of points in a certain direction, usually in the z-direction.

3 Analytical modeling

Assuming that the time used by a computation is dominated by its data traffic, and that a GPU is good at simultaneously scheduling various data movements to avoid unnecessary stalls, we propose the following model to estimate the execution time of a program.

$$\text{Time usage} \geq \max\left(\frac{V_{\text{GM}}}{BW_{\text{GM}}}, \frac{V_{\text{L2}}}{BW_{\text{L2}}}, \frac{V_{\text{SMX}}}{BW_{\text{SMX}}}\right), \tag{2}$$

where V_{GM} denotes the total volume of data loaded from and stored to the global memory, while BW_{GM} denotes its realistically achievable bandwidth. The definitions of V_{L2}, BW_{L2}, V_{SMX} and BW_{SMX} are similar. To predict the actual execution time usage, it is important to accurately quantify the three volumes of data traffic: V_{GM}, V_{L2} and V_{SMX}. We remark that (2) is applicable to all data-traffic dominated computations. The main focus of this paper is on low-order 3D stencil computations, for which we will derive detailed formulas of the three data-traffic volumes. It should also be noticed that not all the stencil codes are definitely bandwidth limited. For some high-order and computation intensive stencils, the bottleneck may be the computation or both computation and memory accessing. In this situation, the time usage of the arithmetic instructions should be considered. Nevertheless, in this paper, we focus on memory bound stencil applications.

3.1 Between on-SMX storage and registers

In (2), V_{SMX} denotes the total volume of data traffic between on-SMX storage and registers. This volume consists of two parts: data loaded from on-SMX storage and data stored into on-SMX storage. For stencil computations, we can safely assume that the number of loads is the same for all threads, so is the number of stores. Moreover, since different threads do not share their registers, the data-traffic volume V_{SMX} can be calculated as:

$$V_{\text{SMX}} = \#\text{threads} \times \left(n^{\text{ld}}_{\text{SMX},T} + n^{\text{st}}_{\text{SMX},T}\right) \times \texttt{sizeof(type)}, \tag{3}$$

where $n^{\text{ld}}_{\text{SMX},T}$ denotes the number of values loaded *per thread* from the on-SMX storage to its registers. Similarly, $n^{\text{st}}_{\text{SMX},T}$ denotes the number of values stored per thread. For double-precision computations, `sizeof(type)` is 8 bytes.

In connection with stencil computations, $n^{\text{st}}_{\text{SMX},T}$ typically equals the number of values that each thread is responsible for updating. For example, for the baseline implementation of 7PT-1 shown in Sect. 2.2, the value of $n^{\text{st}}_{\text{SMX},T}$ is 1. Determining the value of $n^{\text{ld}}_{\text{SMX},T}$, however, requires some care. One important factor is whether the data values to be loaded by a warp align with a 128-byte boundary (an L1 cache line). The load operations thus have two types: aligned and misaligned, where the latter type requires additional data traffic. The following formula calculates the actual number of loads per thread:

$$n^{\mathrm{ld}}_{\mathrm{SMX},T} = \#\text{aligned loads}_T + 2 \times \#\text{misaligned loads}_T. \tag{4}$$

Again, let us take the baseline implementation of 7PT-1 as example. Loads of $u^{\mathrm{old}}_{i,j,k}$, $u^{\mathrm{old}}_{i,j\pm1,k}$, and $u^{\mathrm{old}}_{i,j,k\pm1}$ are typically aligned, whereas loads of $u^{\mathrm{old}}_{i\pm1,j,k}$ are not. Therefore, the value of $\#\text{aligned loads}_T$ is 5 and that of $\#\text{misaligned loads}_T$ is 2. The value of $n^{\mathrm{ld}}_{\mathrm{SMX},T}$ is thus 9.

3.2 Between L2 cache and on-SMX storage

It is reasonable to assume that the CUDA thread blocks are scheduled to the SMXs in a cyclic order [2]. The probability of adjacent thread blocks being assigned to the same SMX is thus very low. We therefore assume that the thread blocks designated to the same SMX do not share data in the on-SMX storage. Consequently, we can use a thread block as the basic granularity to calculate V_{L2} that is needed in (2):

$$V_{\mathrm{L2}} = \#\text{thread blocks} \times \left(n^{\mathrm{ld}}_{\mathrm{L2},B} + n^{\mathrm{st}}_{\mathrm{L2},B}\right) \times \texttt{sizeof(type)}, \tag{5}$$

where $n^{\mathrm{ld}}_{\mathrm{L2},B}$ denotes the number of data values loaded *per thread block* from L2 into on-SMX storage. Similarly, $n^{\mathrm{st}}_{\mathrm{L2},B}$ denotes the number of values stored per thread block.

The number of thread blocks can be easily calculated by dividing the total number of threads by #threads per block, where the latter is decided by the programmer. Determining $n^{\mathrm{st}}_{\mathrm{L2},B}$ is equally straightforward by multiplying $n^{\mathrm{st}}_{\mathrm{SMX},T}$ with #threads per block. However, care is needed when deriving a formula for $n^{\mathrm{ld}}_{\mathrm{L2},B}$, because on-SMX storage capacity miss should be considered. That is, data loaded from L2 cache consists of two parts: the compulsorily required values for each thread block $n^{\mathrm{ld,net}}_{\mathrm{L2},B}$ and the reloaded values due to on-SMX storage capacity miss. Moreover, $n^{\mathrm{ld,net}}_{\mathrm{L2},B}$ needs to cover the computational points processed by one thread block, plus halo regions if applicable. Take for instance the values that need to be loaded for one 7-point stencil. If each CUDA thread is responsible for one computational point, $n^{\mathrm{ld,net}}_{\mathrm{L2},B}$ can be calculated as follows:

$$\begin{aligned} n^{\mathrm{ld,net}}_{\mathrm{L2},B} = {} & B_x \times B_y \times B_z + \big((B_x \times B_z) + (B_x \times B_y)\big) \times halo\_width \\ & + \left(\frac{\text{SMX line size}}{\texttt{sizeof(type)}} \times (B_y \times B_z)\right) \times 2, \end{aligned} \tag{6}$$

where B_x, B_y and B_z denote the dimension of a thread block. The first term $(B_x \times B_y \times B_z)$ of (6) corresponds to the computational points processed. The remaining terms correspond to the halo regions in the three directions, while the value of $halo\_width$ is 2 for stencils to be used in this paper. Here, we remark that the contribution due to the halo region in the x-direction, i.e., the last term in (6), is calculated in a special way. This is because for each halo point in the x-direction, one entire cache line in the on-SMX storage needs to load data from the L2 cache. We can also mention that (6)

remains exactly the same for the case of a 19-point stencil, provided that each CUDA thread is still responsible for one computational point. For the case of chunking, i.e., each CUDA thread is responsible for multiple computational points, the above formula needs to be modified, as will be discussed in Sect. 3.5. Nevertheless, the modeling methodology remains unchanged.

To quantify the number of reloaded values due to on-SMX storage capacity miss, we can incorporate an *SMX miss ratio* when calculating the value of $n^{\text{ld}}_{\text{L2},B}$. More specifically, we propose

$$n^{\text{ld}}_{\text{L2},B} = n^{\text{ld,net}}_{\text{L2},B} \times (1 + \text{SMX miss ratio}), \tag{7}$$

$$\text{SMX miss ratio} = \frac{\text{occupancy} \times \#\text{maximum threads per SMX} \times n^{\text{ld,net}}_{\text{L2},B}}{\#\text{threads per block} \times \frac{\text{on-SMX storage size}}{\texttt{sizeof(type)}}} \times \delta. \tag{8}$$

The rationale behind the above formula of SMX miss ratio is that the more thread blocks that are simultaneously active, the more likely are misses in the on-SMX storage. The value of occupancy can be calculated according to the number of required registers per thread and the shared memory requirement per thread block. Both of them can be obtained by adding option '`--ptxas-options=-v`' during compilation. Moreover, δ is a prescribed small constant in (8), which connects the capacity miss ratio with the total data volume needed by the active thread blocks in the same SMX. One way is to use a large amount of measurements to statistically fit the value of δ, which we consider to be dependent on the GPU hardware configuration, but *independent* of particular stencil computations.

3.3 Between global memory and L2 cache

To quantify the data traffic volume V_{GM} that is needed in (2), we now use as the basic granularity an entire *group* of simultaneously active thread blocks. This is because the L2 cache is shared across all active thread blocks. We assume that the whole computation is divided into several such groups. Since the size of L2 cache is relatively small, data reuse between different groups is negligible. Following the same path of reasoning as above, we can calculate V_{GM} as follows:

$$V_{\text{GM}} = \#\text{groups} \times \left(n^{\text{ld}}_{\text{GM},G} + n^{\text{st}}_{\text{GM},G}\right) \times \texttt{sizeof(type)}, \tag{9}$$

where $n^{\text{ld}}_{\text{GM},G}$ denotes the number of data values loaded *per group* from global memory to L2 cache, whereas $n^{\text{st}}_{\text{GM},G}$ denotes the number of values stored to global memory per group. The value of #groups is calculated by

$$\#\text{groups} = \left\lceil \frac{\#\text{thread blocks}}{\#\text{thread blocks per group}} \right\rceil, \tag{10}$$

where the number of thread blocks per group can be calculated according to the occupancy and the number of SMXs.

As before, calculating $n^{\text{st}}_{\text{GM},G}$ for (9) is straightforward. To calculate $n^{\text{ld}}_{\text{GM},G}$, we use a similar strategy as for $n^{\text{ld}}_{\text{L2},B}$, i.e., dividing it into the compulsorily required values per group $n^{\text{ld,net}}_{\text{GM},G}$ and the reloaded values due to L2 cache capacity miss:

$$n^{\text{ld}}_{\text{GM},G} = n^{\text{ld,net}}_{\text{GM},G} \times (1 + \text{L2 miss ratio}). \tag{11}$$

It is reasonable to assume that thread blocks belonging to the same group are consecutive, if thread blocks are allocated to the SMXs cyclically [2]. Therefore, the data processed by each group will form a continuous big block, which we denote by *subdomain*. The size of a subdomain equals to $n^{\text{ld,net}}_{\text{GM},G}$, which for a 7-point or 19-point stencil can be calculated by

$$n^{\text{ld,net}}_{\text{GM},G} = \left(N_x + \frac{\text{L2 cache line size}}{\texttt{sizeof(type)}} \times 2\right) \times width\_y \times height\_z. \tag{12}$$

Since each group contains many thread blocks, it is reasonable to assume that a subdomain covers all the N_x computational points plus also the halo points in the x-direction. This explains the first term in (12). Moreover, $width\_y$ and $height\_z$ represent the width of a subdomain in the y-direction and its height in the z-direction, respectively,

$$width\_y = B_y \times \left\lceil \frac{\text{\#thread blocks per group} \times B_x}{N_x} \right\rceil + halo\_width, \tag{13}$$

$$height\_z = B_z \times \left\lceil \frac{\text{\#thread blocks per group}}{\frac{N_x \times N_y}{B_x \times B_y}} \right\rceil + halo\_width. \tag{14}$$

In (14), $\frac{N_x \times N_y}{B_x \times B_y}$ calculates the number of thread blocks on the xy-plane. Similar to the capacity miss ratio of on-SMX storage, we can estimate the L2 capacity miss ratio as follows:

$$\text{L2 miss ratio} = \frac{n^{\text{ld,net}}_{\text{GM},G} \times \texttt{sizeof(type)}}{\text{L2 cache size}} \times \epsilon, \tag{15}$$

where ϵ is another prescribed small constant, similar to δ used in (8).

3.4 An illustrating example

To demonstrate the use of the formulas developed so far, let us revisit the baseline CUDA implementation of stencil 7PT-1 shown in Sect. 2.2. The representative NVIDIA K20 GPU is chosen as the hardware testbed, see Table 1 for its specifications. For a specific computation size $N_x = N_y = N_z = 256$ and a particular thread block dimension $B_x = 32$, $B_y = 4$, and $B_z = 1$, Table 2 shows, step by step, how to calculate V_{SMX}, V_{L2} and V_{GM}. For quantifying the latter two volumes we have adopted $\delta = \epsilon = 0.01$. The three data-traffic volumes are thus quantified as 1.25 GB, 844 MB and 534 MB. Compared with the actual measurements provided by `nvprof` [18], the

Table 1 Hardware specifications of a NVIDIA K20 GPU

Items	Values	Items	Values
#SMXs	13	RODC per SMX	48 KB
#SPs	2496	L2 cache size	1280 KB
Clock	0.71 GHz	Measured BW_{SMX}	1215.35 GB/s
Registers per SMX	65536	Measured BW_{L2}	367.87 GB/s
Peak DP rate	1170 Gflop/s	Measured BW_{GM}	160.88 GB/s

accuracy of V_{SMX} is 100 %, whereas for V_{L2} and V_{GM} the accuracy is 94.1 and 96.8 %, respectively.

3.5 Extensions

As mentioned in Sect. 3.2, Formula (6) assumes that each CUDA thread is responsible for one computational point, i.e., following the baseline programming approach. To show that the same modeling methodology is also applicable for the case of multiple computational points per CUDA thread, let us consider for instance a special variant of the 3D blocking technique. More specially, each CUDA thread is now assigned to compute $chunk\_z$ points in the z-direction.

The motivation of doing this 3D blocking is to reduce the data-traffic volumes. We can start by observing that the total number of threads becomes $(N_x \times N_y \times N_z)/chunk\_z$. For the 7PT-1 stencil, we can count that

$$\#\text{aligned loads}_T = 3 \times chunk\_z + 2, \quad \text{and} \quad \#\text{misaligned loads}_T = 2 \times chunk\_z.$$

Averaging to each point, the number of aligned loads is about 3. Compared with the original value of #aligned loads_T for the baseline implementation, which is 5, we can thus see that about 40 % of the aligned loads can now be saved. Consequently, the new value of V_{SMX} is considerably lower.

To see that saving also arises with respect to the data traffic from L2 cache to on-SMX storage, we can derive a new formula of $n_{\mathrm{L2},B}^{\mathrm{ld,net}}$ that incorporates $chunk\_z$ as follows:

$$\begin{aligned} n_{\mathrm{L2},B}^{\mathrm{ld,net}} &= B_x \times B_y \times B_z \times chunk\_z \\ &\quad + \left(B_x \times B_z \times chunk\_z + B_x \times B_y\right) \times halo\_width \\ &\quad + \left(\frac{\text{SMX line size}}{\texttt{sizeof(type)}} \times B_y \times B_z \times chunk\_z\right) \times 2. \end{aligned} \tag{16}$$

Comparing between (6) and (16), we can see that the increase factor in $n_{\mathrm{L2},B}^{\mathrm{ld,net}}$ is less than $chunk\_z$. There is therefore a reduction in the data-traffic volume V_{L2}, due to blocking in the z-direction.

Table 2 Quantifying the data traffic volumes on a K20 GPU for the baseline CUDA implementation of the 7PT-1 stencil, where we have used $N_x = N_y = N_z = 256$, $B_x = 32$, $B_y = 4$, $B_z = 1$, and $\delta = \epsilon = 0.01$

	Model parameter	Source	Value
1	#Maximum threads per SMX	Table 1	2048
2	On-SMX storage size	Table 1	48 KB (used for read-only data cache)
3	#SMXs	Hardware specification Table 1	13
4	L2 cache size	Hardware specification Table 1	1.25 MB
5	L2 cache line size	Hardware specification	32 bytes
6	#Threads per block	$B_x \times B_y \times B_z$	$32 \times 4 \times 1 = 128$
7	#Threads	$N_x \times N_y \times N_z$	$256 \times 256 \times 256 = 16{,}777{,}216$
8	#Thread blocks	$\frac{\#\text{Threads}}{\#\text{Threads per block}}$	$\frac{16{,}777{,}216}{128} = 131{,}072$
9	Occupancy	Program guide [17]	1.0
10	#Thread blocks per SMX	Program guide [17]	$\frac{2048}{128} = 16$
11	#Thread blocks per group	#Thread blocks per SMX × #SMXs	$16 \times 13 = 208$
12	#Aligned loads$_T$	Code analysis	5
13	#Misaligned loads$_T$	Code analysis	2
14	$\#n^{\text{ld}}_{\text{SMX},T}$	Formula (4)	$5 + 2 \times 2 = 9$
15	$\#n^{\text{st}}_{\text{SMX},T}$	Code analysis	1
16	$n^{\text{ld,net}}_{\text{L2},B}$	Formula (6)	$128 + \left(\frac{256}{8} \times 4 + 32 + 128\right) \times 2 = 704$
17	SMX miss ratio	Formula (8)	$\frac{1.0 \times 2048 \times 704}{\frac{128 \times 48 \times 1024}{8}} \times \delta = 1.83\delta$
18	$n^{\text{ld}}_{\text{L2},B}$	Formula (7)	$704 \times (1 + 1.83\delta)$
19	$n^{\text{st}}_{\text{L2},B}$	#Threads per block × $\#n^{\text{st}}_{\text{SMX},T}$	$128 \times 1 = 128$
20	#Groups	Formula (10)	$\left\lceil \frac{131{,}072}{208} \right\rceil = 631$
21	Width_y	Formula (13)	$4 \times \left\lceil \frac{208 \times 32}{256} \right\rceil + 2 = 106$
22	Height_z	Formula (14)	$1 \times \left\lceil \frac{208}{\frac{256 \times 256}{32 \times 4}} \right\rceil + 2 = 3$
23	$n^{\text{ld,net}}_{\text{GM},G}$	Formula (12)	$\left(256 + \frac{32}{8} \times 2\right) \times 106 \times 3 = 83{,}952$
24	L2 miss ratio	Formula (15)	$\frac{83{,}952 \times 8}{1.25 \times 1024^2} \times \epsilon = 0.512\epsilon$
25	$n^{\text{ld}}_{\text{GM},G}$	Formula (11)	$83{,}952 \times (1 + 0.512\epsilon)$
26	$n^{\text{st}}_{\text{GM},G}$	#Thread blocks per group × $n^{\text{st}}_{\text{L2},B}$	$208 \times 128 = 26624$
27	V_{SMX}	Formula (3)	$\frac{16{,}777{,}216 \times (9+1) \times 8}{1024^3} = 1.25$ GB

Table 2 continued

	Model parameter	Source	Value
28	V_{L2}	Formula (5)	$\frac{131{,}072\times(704\times(1+1.83\times0.01)+128)\times8}{1024^2}$ $= 844$ MB
29	V_{GM}	Formula (9)	$\frac{631\times(83{,}952\times(1+0.512\times0.01)+26{,}624)\times8}{1024^2}$ $= 534$ MB

One negative consequence, however, will arise when the size of $chunk\_z$ is so large that

$$\#\text{thread blocks per SMX} \times n_{\mathrm{L2},B}^{\mathrm{ld,net}} \times \texttt{sizeof(type)}$$

exceeds the capacity of on-SMX storage. It is then necessary to adjust the SMX miss ratio as follows:

$$\begin{aligned}\text{SMX miss ratio} = {} & \frac{\text{occupancy} \times \#\text{maximum threads per SMX} \times n_{\mathrm{L2},B}^{\mathrm{ld,net}}}{\#\text{threads per block} \times \dfrac{\text{on-SMX storage size}}{\texttt{sizeof(type)}}} \\ & \times \frac{1 + halo\_width}{chunk\_z + halo\_width} \times \delta \\ & + \log 2 \left(\frac{chunk\_z \times \#\text{warps per thread block}}{\dfrac{\#\text{streaming processors per SMX}}{\#\text{threads per warp}}} \right) \times \eta. \quad (17)\end{aligned}$$

The rationale behind the above formula is that we consider two phases: initial stage and reuse stage. During the initial stage, the on-SMX storage capacity miss ratio complies with Formula (8). At the reuse stage, because $u_{i,j,k-1}^{\mathrm{old}}$ and $u_{i,j,k}^{\mathrm{old}}$ have already been loaded into on-SMX storage, we thus increment Formula (8) with a second term in (17). The purpose is to describe the capacity miss ratio caused by increased data (due to $chunk\_z$) and the consequent warp scheduling, where η is a prescribed small constant.

3.6 Remarks

Our analytical model has not considered shared memory of the on-SMX storage. This is because the performance of implementations using read-only data cache (RODC) is better than that of shared memory-based implementations, as long as no time tiling is applied between sweeps of a 3D stencil. There are, however, no principal obstacles to applying the same modeling methodology to CUDA implementations that also make use of shared memory.

In the case of very low occupancy, there may not be enough warps to hide the memory latency. Estimates of the execution time that are produced by our model will thus be too optimistic. The capacity miss ratio formulas are rather simplistic, which

may also cause inaccurate time estimates, the inaccuracy of the model, especially for a very large value of *chunk_z*. However, for moderate sizes of *chunk_z*, experiments show that the impact of the capacity miss ratios is insignificant.

4 Validation

4.1 Experimental setup

An NVIDIA K20 GPU is used as the hardware testbed, with its specifications shown in Table 1. For the Kepler architecture, each SMX has 48 KB of RODC. As explained in Sect. 3.6, on-SMX storage in this paper only considers RODC, not shared memory. For measuring the realistic bandwidths BW_{SMX} and BW_{L2}, we have designed micro benchmarks similar to those published in [20]. The realistic value of BW_{GM} is measured using NVIDIA's `bandwidthTest` program.

In addition to the 7PT-1 stencil from Sect. 2.2, we will use three more representative low-order 3D stencils to validate the accuracy of our modeling methodology presented in the previous section. Two different stencil shapes, 7-point and 19-point, are associated with these stencils, which also differ in the number of input arrays and how these arrays are accessed.

$$u^{\mathrm{new}}_{i,j,k} = \alpha u^{\mathrm{old}}_{i,j,k} + \beta \left(u^{\mathrm{old}}_{i\pm1,j,k} + u^{\mathrm{old}}_{i,j\pm1,k} + u^{\mathrm{old}}_{i,j,k\pm1} \right) \quad \text{(7PT-1)}$$

$$u^{\mathrm{new}}_{i,j,k} = \alpha \gamma_{i,j,k} + \beta \left(u^{\mathrm{old}}_{i\pm1,j,k} + u^{\mathrm{old}}_{i,j\pm1,k} + u^{\mathrm{old}}_{i,j,k\pm1} \right) \quad \text{(7PT-2)}$$

$$\begin{aligned} u^{\mathrm{new}}_{i,j,k} = {} & u^{\mathrm{old}}_{i,j,k} + \gamma_{i,j,k} \\ & + \alpha \Big(\left(\kappa_{i+1,j,k} + \kappa_{i,j,k} \right) \left(u^{\mathrm{old}}_{i+1,j,k} - u^{\mathrm{old}}_{i,j,k} \right) \\ & - \left(\kappa_{i,j,k} + \kappa_{i-1,j,k} \right) \left(u^{\mathrm{old}}_{i,j,k} - u^{\mathrm{old}}_{i-1,j,k} \right) \\ & + \left(\kappa_{i,j+1,k} + \kappa_{i,j,k} \right) \left(u^{\mathrm{old}}_{i,j+1,k} - u^{\mathrm{old}}_{i,j,k} \right) \\ & - \left(\kappa_{i,j,k} + \kappa_{i,j-1,k} \right) \left(u^{\mathrm{old}}_{i,j,k} - u^{\mathrm{old}}_{i,j-1,k} \right) \\ & + \left(\kappa_{i,j,k+1} + \kappa_{i,j,k} \right) \left(u^{\mathrm{old}}_{i,j,k+1} - u^{\mathrm{old}}_{i,j,k} \right) \\ & - \left(\kappa_{i,j,k} + \kappa_{i,j,k-1} \right) \left(u^{\mathrm{old}}_{i,j,k} - u^{\mathrm{old}}_{i,j,k-1} \right) \Big) \end{aligned} \quad \text{(7PT-3)}$$

$$\begin{aligned} u^{\mathrm{new}}_{i,j,k} = {} & \alpha \gamma_{i,j,k} + \beta \left(u^{\mathrm{old}}_{i\pm1,j,k} + u^{\mathrm{old}}_{i,j\pm1,k} + u^{\mathrm{old}}_{i,j,k\pm1} \right) \\ & + \left(u^{\mathrm{old}}_{i\pm1,j\pm1,k} + u^{\mathrm{old}}_{i\pm1,j,k\pm1} + u^{\mathrm{old}}_{i,j\pm1,k\pm1} \right). \end{aligned} \quad \text{(19PT)}$$

Four different computation sizes 64^3, 128^3, 256^3 and 512^3 are tried for all the four stencils, each with a baseline (one-thread-one-point) CUDA implementation and another CUDA implementation that adopts blocking in the z-direction. All the computations have used double precision.

To limit the total number of experiments, we have restricted the thread block configurations as follows: the value of B_x must be divisible by 32 and the thread block size is within 1024. We have used NVIDIA's `nvprof` tool to measure the actual data traffic volumes. All the CUDA implementations are compiled with `nvcc`.

4.2 Validation using baseline implementations

We first study the accuracy of V_{SMX}, V_{L2} and V_{GM}, which are quantified by Formulas (3), (5) and (9), by comparing them with actual measurements obtained by `nvprof`. The four baseline CUDA implementations are investigated, for which we have varied the computation size and the thread block configuration. For a given 3D stencil, the value of V_{SMX} only depends on the computation size, independent of the thread block configuration. This is indeed confirmed by the actual measurements from `nvprof`, with 100 % accuracy. For V_{L2} and V_{GM}, Table 3 shows their average prediction errors. We can observe a good accuracy for both data traffic volumes. The average predicted error ranges from 0 to 9 %, indicating that our modeling methodology presented in Sect. 2 works well for all the four baseline CUDA implementations.

Figure 1 shows, for the baseline implementation of 7PT-1 and computation size 256^3, a detailed comparison of predicted values of V_{SMX}, V_{L2} and V_{GM} against the actual measurements from `nvprof`. It can be seen that, the predicted data-traffic volumes closely follow the actual measurements for all three memory connections,

Table 3 Comparing actual measurements of V_{L2} and V_{GM} with their predictions for four baseline implementations (using $\delta = \epsilon = 0.01$)

Size	Average error in V_{L2}				Average error in V_{GM}			
	64^3 (%)	128^3 (%)	256^3 (%)	512^3 (%)	64^3 (%)	128^3 (%)	256^3 (%)	512^3 (%)
7PT-1	3.60	4.00	3.45	3.45	0.86	2.00	1.72	1.59
7PT-2	2.78	3.25	2.75	2.73	0.18	1.01	1.41	1.32
7PT-3	6.97	6.82	6.88	4.46	1.12	3.81	1.58	1.90
19PT	8.43	8.14	8.70	8.76	0.39	0.54	0.04	0.00

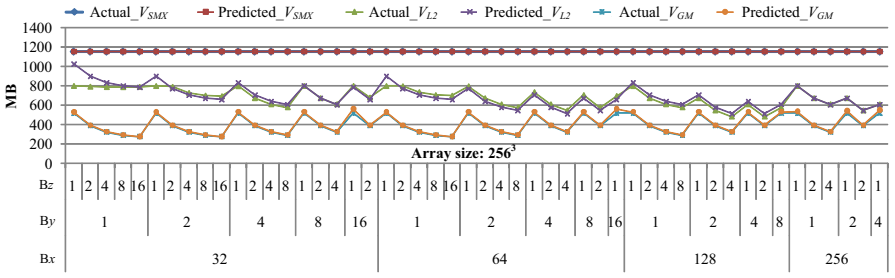


Fig. 1 A detailed comparison of predicted values of V_{SMX}, V_{L2} and V_{GM} against actual measurements, for the baseline implementation of stencil 7PT-1 with computation size 256^3

except for small thread block sizes. In particular, for block size of 32 or 64, the predicted V_{L2} values are distinctly higher than the actual measurements. This discrepancy is probably due to a certain level of data sharing between thread blocks through RODC, which is not considered in Formula (5). Another interesting observation is that V_{GM} decreases with an increasing value of B_z, which implies that our assumption about the group of simultaneously active thread blocks is reasonable for CUDA programming.

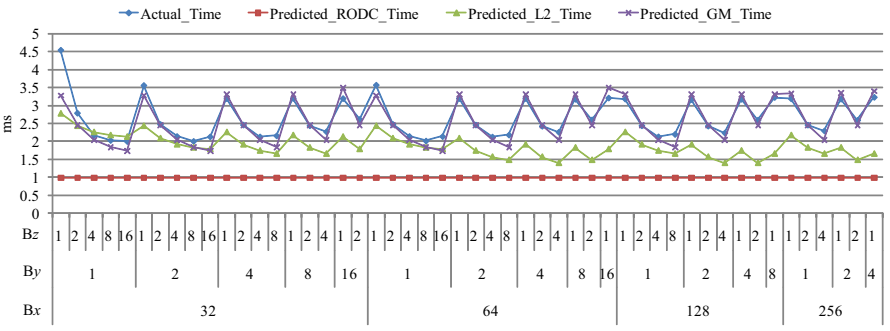
We have also experimented with ignoring L2 cache capacity misses, by setting $\epsilon = 0$ in (15). It turns out to have a very small effect on the accuracy of predicted V_{GM} values. This is because the amount of data required by one group is smaller than the size of L2 cache on K20, see Table 2.

Once predictions of V_{SMX}, V_{L2} and V_{GM} are ready, we can use Formula (2) and bandwidths given in Table 1 to predict the execution time usages. Figure 2 shows a comparison between the actual time usages and the predictions for the four baseline CUDA implementations using computation size 256^3. The four plots show that the predicted time usages due to V_{GM}/BW_{GM} (or V_{L2}/BW_{L2}) closely follow the actual time measurements. When the thread block size is smaller than 64, the predicted time usages are noticeably lower than the actual measurements. This is because occupancy can be quite low for small thread blocks, such as 0.25 for thread block size of 32. Consequently, low occupancy may result in that the GPU being partially idle. The performance bottleneck for stencils 7PT-1, 7PT-2 and 7PT-3 is the data traffic between global memory and L2 cache. For the baseline implementation of stencil 19PT, the performance bottleneck can be between L2 cache and on-SMX storage (RODC) for small values of B_y or B_z. This is because the data reuse ratio within a thread block can be very low, thereby large values of V_{L2}. We can also see that accuracy of time usage prediction for stencil 19PT is lower than that of the three 7-point stencils.

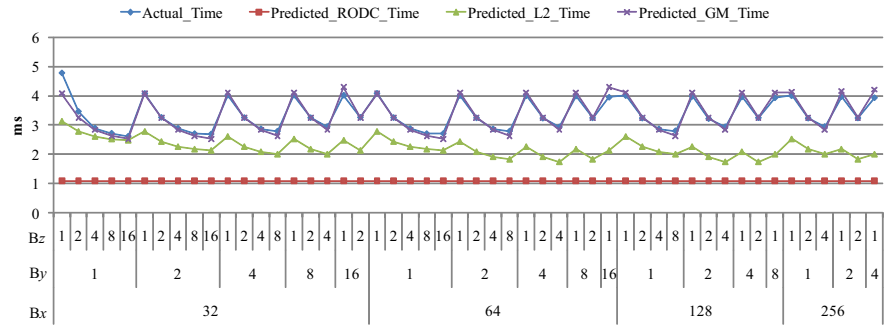
4.3 Validation using z-blocked implementations

Here, we continue to validate our modeling methodology for non-baseline CUDA implementations of 3D stencil computations. As discussed in Sect. 3.5, we now investigate another set of implementations of the four stencils that allow each CUDA thread to compute $chunk\_z$ points. Table 4 compares the predicted values of V_{L2} and V_{GM} against the actual measurements provided by `nvprof`. We have varied the values of B_x and B_y (while fixing B_z at 1) together with varying $chunk\_z = 2^i, 0 \leq i \leq 8$. The accuracy of V_{L2} predictions is quite good for the three 7-point stencils, whereas that of 19PT is less satisfactory. We believe this is due to an inaccurate quantification of the SMX miss ratio, i.e., Formula (17), for this 19-point stencil. For V_{GM}, the average prediction accuracy is 96.5 %. The error is mainly caused by large values of $chunk\_z$, the accuracy of predicted V_{GM} values is thus lower for computation size 512^3 than that of other computation sizes.

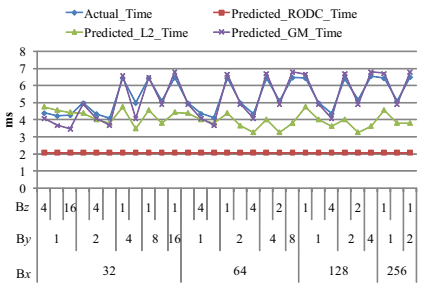
Similarly, we give a detailed comparison between the predicted data-traffic volumes and actual measurements in Fig. 3, for stencil 7PT-1 and computation size 256^3. Only results with $B_x = 32$ are shown in the plot. Different from the baseline implementation, blocking in the z-direction considerably improves data reuse in registers. The predicted value of V_{SMX} thus decreases when $chunk\_z$ becomes larger. We can also see a perfect



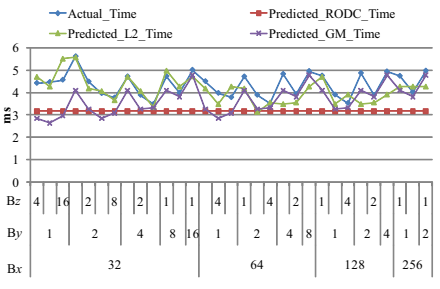
(a) 7PT-1



(b) 7PT-2



(c) 7PT-3



(d) 19PT

Fig. 2 Comparing actual time usages with predictions that are given by Formula (2), for four baseline implementations with computation size 256^3, B_x, B_y, and B_z are the thread block sizes in the corresponding directions, $Actual\_Time$ donates the actual program time, while $Predicted\_RODC\_Time$ means the estimated time according to data traffic volume of on-SMX storage, $Predicted\_L2\_Time$ and $Predicted\_GM\_Time$ are similar

accuracy of the V_{SMX} predictions. The predicted values of V_{L2} and V_{GM} also closely follow the actual measurements.

In Fig. 4, we also study the impact of L2 cache capacity miss on the accuracy of V_{GM} predictions. That is, an additional set of V_{GM} predictions (green curve) is produced by setting $\epsilon = 0$ in Formula (15). It can be seen that when $chunk\_z \leq 32$, omitting

Table 4 Comparing actual measurements of V_{L2} and V_{GM} with their predictions for four z-blocked implementations

Size	Average error in V_{L2}				Average error in V_{GM}			
	64^3 (%)	128^3 (%)	256^3 (%)	512^3 (%)	64^3 (%)	128^3 (%)	256^3 (%)	512^3 (%)
7PT-1	5.93	6.41	5.79	5.82	3.41	2.03	3.76	8.58
7PT-2	5.22	5.93	7.12	7.27	3.11	2.60	4.49	8.56
7PT-3	4.91	5.49	6.04	6.11	4.37	3.18	4.31	9.87
19PT	11.50	10.71	12.54	12.59	2.43	2.03	3.95	6.91

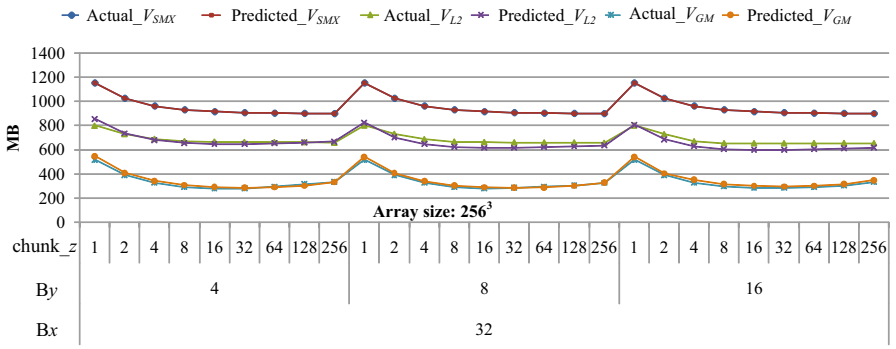
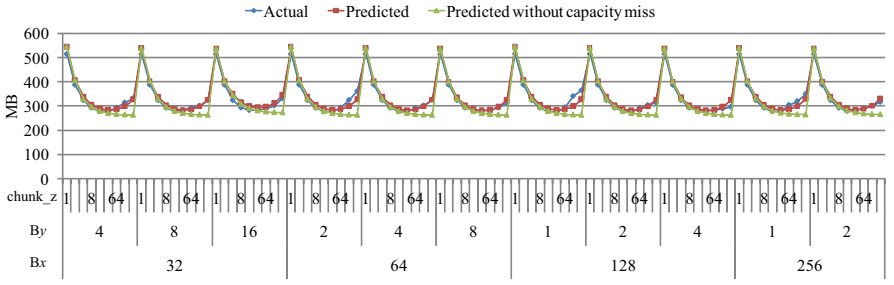


Fig. 3 A detail comparison of predicted values of V_{SMX}, V_{L2} and V_{GM} against actual measurements, for a z-blocked implementation of stencil 7PT-1 with computation size 256^3

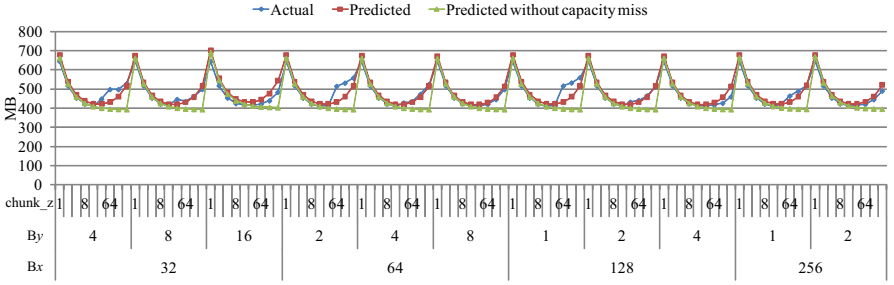
L2 cache capacity miss has a very small impact on the accuracy of V_{GM} predictions. However, when *chunk_z* increases from 32, the actual amount of data loaded from GPU's global memory also increases. This is due to the increased capacity miss of L2 cache. It is therefore very important to use a nonzero ϵ value associated with the largest *chunk_z* value of 256. Otherwise, omitting ϵ can reduce the prediction accuracy of V_{GM} by as much as 30 %. This applies to all the four stencils.

Figure 5 shows a comparison between the actual time usages and predictions. The red curves arise from Formula (2). It can be seen that, when *chunk_z* is smaller than 16, the prediction accuracy achieves up to 95 %. However, when *chunk_z* becomes larger, although the accuracy of V_{GM} predictions is still very high (see Fig. 4), the predicted time usages (red curves) have a noticeable gap to the actual time usages (blue curves). We believe that, for very large values of *chunk_z*, the number of thread blocks may be so small that will result in an imbalance in scheduling. In addition, a low occupancy rate will also impact the execution time. Therefore, we can adjust the predicted time usage by adopting an imbalance coefficient that is related to the number of groups and the occupancy efficiency:

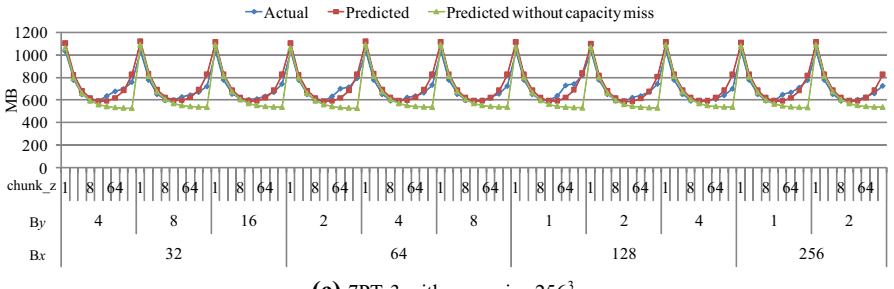
$$Time_{\text{Adjusted}} = Time_{\text{Predicted}} \times \frac{1 + \frac{1}{\#\text{groups}}}{\text{occupancy efficiency}}, \tag{18}$$



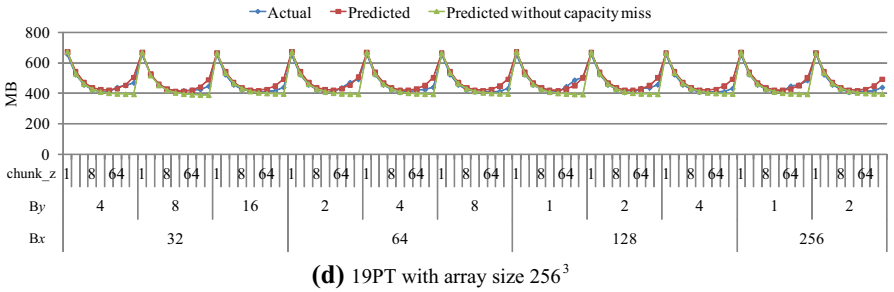
(a) 7PT-1 with array size 256^3



(b) 7PT-2 with array size 256^3



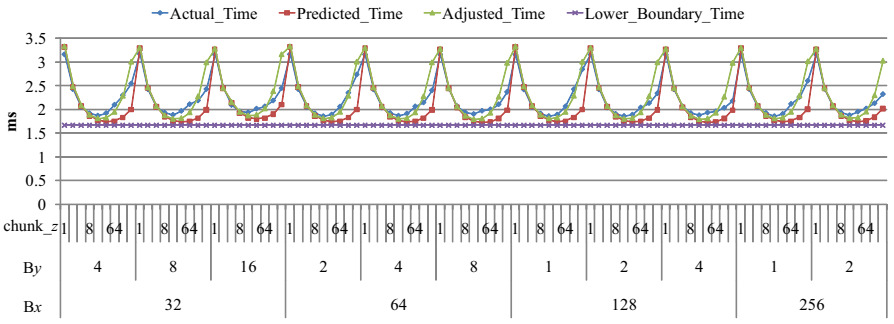
(c) 7PT-3 with array size 256^3



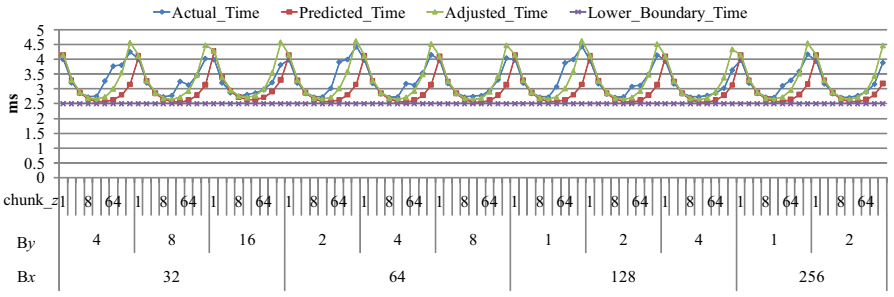
(d) 19PT with array size 256^3

Fig. 4 Comparing actual measurements of V_{GM} against predictions that consider or ignore L2 cache capacity miss (color figure online)

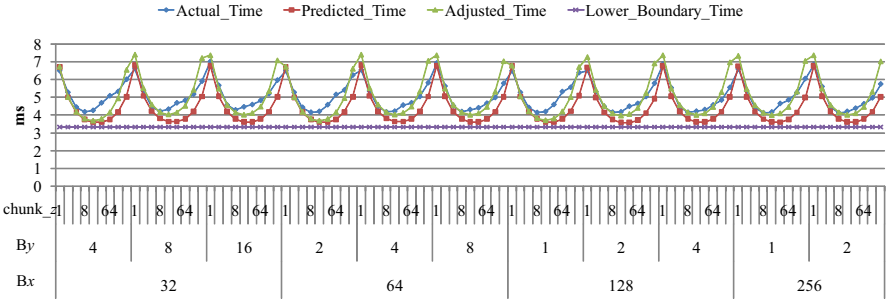
where occupancy efficiency is the ratio between the current occupancy rate and the maximal occupancy. Formula (18) means that when the last group is executed, the hardware efficiency is lower than when other groups are executed, because of too few



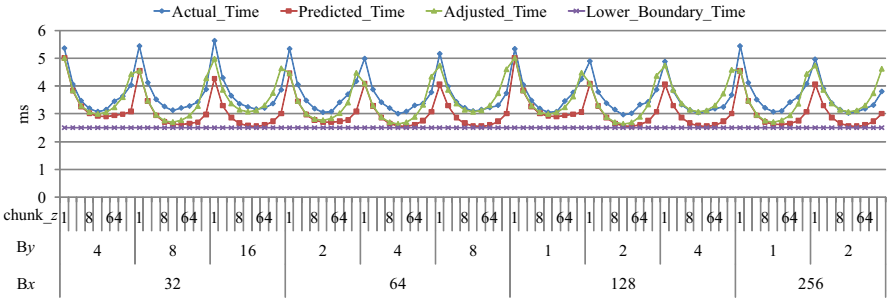
(a) 7PT-1 with array size 256^3



(b) 7PT-2 with array size 256^3



(c) 7PT-3 with array size 256^3



(d) 19PT with array size 256^3

Fig. 5 Comparing the actual time usages with predicted time usages for the four z-blocked implementations. Predictions labeled as "Adjusted" are produced by Formula (18) (color figure online)

thread blocks in the last group. For instance, when the thread block size is 128, the 19PT stencil achieves its maximal occupancy of 0.875. When the thread block size is 512, the actual occupancy rate is reduced to 0.75. The occupancy efficiency is thus $\frac{0.75}{0.875} = 85.7\,\%$. The green curves in Fig. 5 denote the adjusted time usage predictions following Formula (18).

5 Related work

To investigate the performance of stencil computations, earlier work focused on using various optimizations to manually improve the data locality and parallelism on multicore and many-core processors [3,6,10]. We examined in [23,24] the performance of OpenCL-implemented stencil computations on NVIDIA's GPU and compared them with the corresponding results with the CUDA programming model. The results showed that the performance of stencil computation on GPU is mainly determined by the data transfer. Based on these observations, we have proposed the analytical model in this paper.

To relieve the programming burden, automated code generation and tuning methods are widely used to achieve high performance of stencil computations on GPU [7,11]. Mint [25] provided an OpenMP-like method to generate CUDA code of 3D stencil computations. However, the programmer always faced the problem of choosing the best thread block configuration. Zhang and Mueller [27] presented an auto-generator for 3D stencil computation on GPU clusters. Although an auto-tuning method was introduced, they restricted the value of $chunk\_z$ to be the size of array, which may not find the optimal configuration.

There also exist some works that attempt to understand the performance of stencil computations through modeling or quantitative analysis. Rahman et al. [19] proposed a regression analysis model to study the performance of stencil computations on multicore CPUs by using hardware counters to monitor the efficiency of architectural components. This approach requires the programmer to conduct heavy experiments to obtain the statistical metrics. Datta et al. [4] reviewed the cache reuse optimizations of stencil computations on modern microprocessors. They also studied the impact of different blocking sizes on the performance, especially focused on the cache miss ratio of modern processors. Kamil et al. [9] also proposed a performance model to study the performance of stencil computations on modern processors. Very recently, de la Cruz and Araya-Polo [5] studied the performance of stencil computation on modern HPC architectures. They focused on quantifying the cache misses on multicore or many-core architectures. As published, their prediction error for most relevant cases is 5–15 %. Stengel et al. quantified the performance bottlenecks of stencil computations based on the execution-cache-memory model [22]. According to the analysis, they explained the effect of typical optimization approaches. However, their focus has mainly been on the modern multicore processors, for which the hardware and the programming model are quite different from the modern GPU.

Meng and Skadron [12] studied the performance of iterative stencil loops on GPUs with ghost optimizations. Their method focuses on the ghost zone optimization and estimates the overhead of instructions, but not the data throughput. The prediction

error for 3D stencils is 28–30 %. Meanwhile, the size of the array used in their tests is 100^3, allowing most of the elements to be kept in cache during kernel execution.

Regarding other performance modeling works, Williams et al. [26] proposed the Roofline model, which can be used to visualize compute-bounded or memory-bounded multicore architectures. Hong and Kim [8] proposed an MWP-CWP analytical model to estimate the cost of memory operations based on the compiled PTX codes. Based on [8], authors of [21] developed an analysis framework to predict performance potential and provide programmer-interpretable metrics by following the work-depth graph formalism. Based on program dependence graph, Baghsorkhi et al. [1] presented a performance model to assist programming and tuning GPGPU applications. Nugteren et al. proposed a detailed GPU cache model based on reuse distance theory [15]. They extended the reuse distance model to GPU by combining the GPU parallel hierarchies, such as threads, warps and thread blocks, which is similar with the execution granularity of the three memory hierarchies used in our paper.

6 Conclusion

This paper has proposed an analytical model to study the GPU performance of low-order 3D stencil computations. Our focus has been on quantifying the volumes of data traffic at three stages: (1) between registers and the on-SMX storage, (2) between the on-SMX storage and the L2 cache, (3) between the L2 cache and GPU's device memory. The corresponding granularities are a CUDA thread, a thread block, and a group of simultaneously active thread blocks. The quantified data-traffic volumes, together with realistic memory bandwidths, can then be used to predict the execution time usage. Experimental results have shown good accuracy of the three quantified data-traffic volumes, with a satisfactory accuracy of time usage predictions as the result.

Acknowledgments The authors gratefully acknowledge the support from the National Natural Science Foundation of China under NSFC Nos. 61033008, 61103080 and 61272145, SRFDP Nos. 20104307110002 and 20124307130004, Innovation in Graduate School of NUDT Nos. B100603, B120605, the FRINATEK program of the Research Council of Norway under No. 214113/F20.

References

1. Baghsorkhi SS, Delahaye M, Patel SJ, Gropp WD, Hwu WMW (2010) An adaptive performance modeling tool for GPU architectures. In: Proceedings of PPoPP'10. ACM, New York, pp 105–114. doi:10.1145/1693453.1693470
2. Bakhoda A, Yuan GL, Fung WW, Wong H, Aamodt TM (2009) Analyzing cuda workloads using a detailed GPU simulator. In: IEEE international symposium on performance analysis of systems and software (ISPASS'09). IEEE, pp 163–174
3. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K (2008) Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of SC'08. IEEE Press, Piscataway, pp 4:1–4:12. doi:10.1109/SC.2008.5222004
4. Datta K, Kamil S, Williams S, Oliker L, Shalf J, Yelick K (2009) Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Rev 51(1):129–159
5. de la Cruz R, Araya-Polo M (in press) Modeling stencil computations on modern HPC architectures

6. De La Cruz R, Araya-Polo M (2014) Algorithm 942: semi-stencil. ACM Trans Math Softw (TOMS) 40(3):23
7. Holewinski J, Pouchet LN, Sadayappan P (2012) High-performance code generation for stencil computations on GPU architectures. In: Proceedings of ICS'12. ACM, New York, pp 311–320. doi:10.1145/2304576.2304619
8. Hong S, Kim H (2009) An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In: Proceedings of ISCA'09. ACM, New York, pp 152–163. doi:10.1145/1555754.1555775
9. Kamil S, Husbands P, Oliker L, Shalf J, Yelick K (2005) Impact of modern memory subsystems on cache optimizations for stencil computations. In: Proceedings of MSP'05. ACM, New York, pp 36–43. doi:10.1145/1111583.1111589
10. Kamil S, Datta K, Williams S, Oliker L, Shalf J, Yelick K (2006) Implicit and explicit optimizations for stencil computations. In: Proceedings of MSPC'06. ACM, New York, pp 51–60. doi:10.1145/1178597.1178605
11. Kamil S, Chan C, Oliker L, Shalf J, Williams S (2010) An auto-tuning framework for parallel multicore stencil computations. In: Proceedings of IPDPS'10, pp 1–12. doi:10.1109/IPDPS.2010.5470421
12. Meng J, Skadron K (2009) Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In: Proceedings of ICS'09. ACM, New York, pp 256–265. doi:10.1145/1542275.1542313
13. Micikevicius P (2009) 3D finite difference computation on GPUs using CUDA. In: GPGPU-2. ACM, New York, pp 79–84. doi:10.1145/1513895.1513905
14. Nickolls J, Dally W (2010) The GPU computing era. Micro IEEE 30(2):56–69. doi:10.1109/MM.2010.41
15. Nugteren C, van den Braak GJ, Corporaal H, Bal H (2014) A detailed GPU cache model based on reuse distance theory. In: IEEE 20th international symposium on high performance computer architecture (HPCA). IEEE, pp 37–48
16. NVIDIA T (2013) K20-k20x GPU accelerators benchmarks. ApplicationPerformance Technical Brief, Nvidia. http://www.nvidia.com/docs/IO/122874/K20-and-K20X-application-performance-technical-brief.pdf
17. NVIDIA C (2012a) CUDA API reference manual
18. Profiler user's guide.http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf
19. Rahman SMF, Yi Q, Qasem A (2011) Understanding stencil code performance on multicore architectures. In: Proceedings of the 8th ACM international conference on computing frontiers. ACM, New York p 30
20. Schäfer A, Fey D (2011) High performance stencil code algorithms for GPGPUs. Procedia Comput Sci 4:2027–2036
21. Sim J, Dasgupta A, Kim H, Vuduc R (2012) A performance analysis framework for identifying potential benefits in GPGPU applications. In: Proceedings of PPoPP'12. ACM, New York, pp 11–22. doi:10.1145/2145816.2145819
22. Stengel H, Treibig J, Hager G, Wellein G (2014) Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. arXiv:1410.5010
23. Su H, Wu N, Wen M, Zhang C, Cai X (2013a) On the GPU–CPU performance portability of OpenCL for 3D stencil computations. In: International conference on parallel and distributed systems (ICPADS). IEEE, pp 78–85
24. Su H, Wu N, Wen M, Zhang C, Cai X (2013b) On the GPU performance of 3D stencil computations implemented in OpenCL. In: Supercomputing. Springer, New York, pp 125–135
25. Unat D, Cai X, Baden SB (2011) Mint: realizing CUDA performance in 3D stencil methods with annotated C. In: Proceedings of ICS'11. ACM, New York, pp 214–224. doi:10.1145/1995896.1995932
26. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76. doi:10.1145/1498765.1498785
27. Zhang Y, Mueller F (2012) Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In: Proceedings of CGO'12. ACM, New York, pp 155–164. doi:10.1145/2259016.2259037